INTRODUÇÃO À ARQUITETURA E ENGENHARIA JURÍDICA

com Lawtex®

Looplex

Anderson Fraiha Machado, Leilani Dian Mendes, Lucas Nogueira Garcez

Introdução à Arquitetura e Engenharia Jurídica com Lawtex

São Paulo
Janeiro de 2018

Resumo

Este material se propõe a realizar um levantamento a respeito do cenário tecnológico que nutri o ecossistema jurídico no âmbito de automação de documentos. Após definições gerais a respeito do direito computacional, apresenta-se ferramentas essenciais para formação de arquitetos e engenheiros jurídicos. Não só ferramentas de projeto, mas também bases de *design thinking*, métodos ágeis e diagramas técnicos (modelo entidade relacionamentos, árvores de decisão, *mind mapping*, etc) foram apresentados. Os autores ainda dedicaram algum tempo com o intuito de munir o leitor com recursos técnicos básicos sobre algoritmos e fundamentação lógica (teoria dos conjuntos e lógica proposicional), as quais se apresentam como um degrau conceitual no caminho para o mundo da programação propriamente. É um material que vale a pena conferir, realizar os exercícios propostos, e não desistir na primeira notação grega. Logo você verá que sua mente é pura lógica e sua intuição é pura matemática.

Palavras-chave: engenharia jurídica, arquitetura jurídica, Lawtex, automação de documentos.

Sumário

1 Introdução

As inovações tecnológicas no âmbito do Direito não são mais temas debatidos como problemas dos advogados do futuro. Richard Tromans é preciso ao afirmar que a inteligência artificial relacionada ao Direito "...não se trata de uma questão de *e se* ou *quando*" [1], trata-se de uma realidade que está acontecendo, tanto no cenário internacional como no cenário nacional.

O problema central a ser discutido é como os profissionais do Direito podem se preparar para os novos desafios decorrentes dessas inovações, bem como quais habilidades serão necessárias para suprir as novas demandas.

1.1 Considerações Preliminares

Compreender os principais aspectos referentes ao processo de automação e engenharia jurídica para que os alunos possam compreender o cenário em que o curso está inserido.

Evolução tecnológica nas profissões

Inicialmente é importante identificar as alterações que as novas tecnologias impactaram na atuação dos profissionais, apontando os três principais marcos da Revolução Industrial.

Produção Industrial (Meados dos séculos XVIII-XIX)

A industrialização pode ser definida como o processo de modernização em que passam os meios de produção utilizados por uma sociedade.

A Primeira Revolução Industrial proporcionou uma transformação no modo como as relações econômicas e sociais eram desenvolvidas, impactando diretamente na atuação profissional dos trabalhadores.

As atividades que antes eram desenvolvidas pelos trabalhadores de forma artesanal, ou seja, centralizadas em um mesmo produtor responsável pela execução de todas as etapas da produção, desde o preparo da matéria-prima até o acabamento final e passaram pelo processo de industrialização. Esse processo envolvia a utilização de máquinas para auxiliar na execução das etapas, promovendo a escalabilidade na produção.

O processo de industrialização fez aumentar a criação de novas oportunidades de emprego em fábricas e minas extrativistas, estimulando uma forte migração de trabalhadores do campo para as grandes cidades. Assim, o mercado de trabalho era dominado em sua grande maioria por operários, sendo necessária a realização de

suas atividades de maneira segregada e especializada.

A Inglaterra foi o país que se destacou no âmbito da Primeira Revolução Industrial, em razão da invenção da máquina a vapor, que tinha como objetivo aumentar a produtividade e maior exploração do trabalho.

Linha de Produção (Meados século XIX-XX)

Após a deflagração da Primeira Revolução Industrial e o aprimoramento dos meios de produção industriais, passou-se à estruturação e organização de tais meios.

Esta nova forma de produção, conhecida como produção em série ou em massa, era caracterizada pela atuação de operários, com ajuda de máquinas, que se especializavam em diversas funções específicas e repetitivas. Como resultado, verificou-se uma maior eficiência na produção, aumentando a produção e, ao mesmo tempo, reduzindo o tempo e custos de produção dos produtos.

Neste cenário, os trabalhadores se especializavam em funções específicas relacionadas ao manuseio e operações de máquinas.

A produção em massa teve como precursora o sistema de produção desenvolvido por Henry Ford em 1914, conhecido como Fordismo. O Fordismo tinha como objetivos a redução de custos de produção da fábrica de automóveis, reduzindo os custos dos veículos para a venda, atingindo, assim, um maior número de consumidores.

Mecanização (Século XX)

A presente forma de produção foi caracterizada pela utilização de tecnologia, por meio da produção de conhecimento científico, para o desenvolvimento e aprimoramento das etapas que envolvem a produção industrial.

Dessa forma, os avanços da automação e da informática são incorporados junto à cadeia do processo produtivo, dependendo da produção de alta tecnologia e de mão-de-obra cada vez mais qualificada e especializada.

Essa forma de produção exige um profissional que domine as especificidades das sua área de atuação, bem como o manuseio de ferramentas tecnológicas que o auxiliam em suas demandas diárias.

Deste modo, verificando as três fases da Revolução Industrial os profissionais passaram de detentores do conhecimento completo do processo de produção para uma atuação especializada. E mais recentemente, uma atuação especializada com o necessário conhecimento de ferramentas tecnológicas.

Novas habilidades profissionais exigidas no mercado jurídico

A presença recente de algoritmos e inteligência artificial têm provocado alterações na forma como lidamos com nossas relações sociais e profissionais. A utilização de tecnologias disruptivas, ou seja, tecnologias que alteram o modelo tradicional de operação, como exemplo os aplicativos de serviços de transporte por meio de carros particulares substituindo o bom e velho táxi, bem como plataformas de viagem *online* que estão substituindo as agências de turismo, estão levantando diversos questionamentos.

O Direito há anos perpetua um modelo de negócio estático e previsível, em que existem etapas para que o advogado possa chegar ao topo de um escritório. Usualmente o profissional inicia sua carreira como estagiário passando por práticas burocráticas e não especificamente jurídicas, para iniciar seu contato com questões jurídicas de baixa complexidade nos primeiros anos de formação. Em seguida acaba escolhendo os temas que mais lhe atraem e permanece em uma área de atuação específica até o momento que passa a ter muita experiência prática e, consequentemente, um contato maior com os clientes e negociações, atuando não mais exclusivamente como um advogado, mas sim, um administrador.

No entanto, com a utilização de tecnologias disruptivas no âmbito jurídico, toda a organização e estrutura tendem a mudar. Tarefas que antes eram realizadas exclusivamente por estagiários e recém formados (pesquisas de legislação e jurisprudência, elaboração de documentos etc) podem ser realizadas com maior agilidade e padronização com o auxílio de *softwares* jurídicos. Os profissionais que atuam no meio da cadeia passarão a ser cobrados por meio de estatística de produção e qualidade de documentos jurídicos produzidos. E os advogados com mais experiência que normalmente estão no topo da cadeia terão que lidar com a pressão de clientes que querem a redução de custos em razão da utilização de tecnologia no processo de produção de documentos.

Em recente pesquisa realizada pelo grupo Editorial Person em parceria com a fundação britânica Nesta e a Oxford Martin School, denominada "The future of skills: Employment in 2030", afirma-se que o impacto das tecnologias está promovendo mudanças significativas nas relações profissionais, portanto, os profissionais devem ter como prioridade o "desenvolvimento de habilidades e no aprender a aprender" [2].

Nesse novo cenário quais são as habilidades necessárias para acompanhar as alterações que a profissão do Direito está sofrendo? A seguir indicamos algumas sugestões (sugestões, pois a cada instante podem surgir outras) de habilidades que acreditamos serem necessárias para os profissionais que atuam com o Direito:

(a) **Interação entre homem e tecnologia.** Os profissionais que tiverem a habilidade de introduzir e utilizar as diversas tecnologias na prática profissional certamente terão vantagens, como agilidade na prestação de serviços, capacidade de redução de custos e consequentemente ampliar a carteira de clientes;

(b) **Análise e estruturação de informações.** Atualmente grande parte dos advogados estão acostumados com o processo de utilizar minutas padrão e copiar e colar cláusulas ou teses para atender demandas diferentes de clientes. No entanto, como as ferramentas tecnológicas auxiliarão na tarefa burocrática de produzir documentos, caberá ao profissional compreender e mapear todas as informações que precisam ser levantadas para estruturação do documento jurídico. Será necessário um conhecimento mais amplo do Direito e não específico e especializado como vem sendo aplicado;

(c) **Interdisciplinariedade com outras áreas.** Há anos o Direito tem sido ensinado nas universidades e faculdades tradicionais da mesma forma e isso também se reflete na atividade prática do Direito. Poucas são as instituições que enfatizam o conhecimento de outras áreas muito importantes para os profissionais da área jurídica como contabilidade, finanças, administração de empresas, estatística, ciências da computação, dentre outras. O profissional do Direito deve estar preparado para dialogar com outras áreas e até mesmo se utilizar de teorias e ferramentas de outras áreas para aprimorar a prestação de seus serviços.

Desta forma, para que os profissionais que atuam com o Direito não se tornem profissionais obsoletos, independentemente de sua atuação, novas habilidades deverão ser desenvolvidas. Um dos objetivos desse

livro é justamente trazer a consciência aos profissionais do Direito e estimular o desenvolvimento de novas habilidades.

O processo de automação de documentos jurídicos

Conforme verificado, a revolução tecnológica mudou nossa sociedade. Mas ainda não tinha impactado de forma relevante a prestação dos serviços jurídicos. Escritórios de advocacia e o Poder Judiciário continuam a trabalhar essencialmente da mesma forma há décadas.

A habilidade de pensar problemas jurídicos e comunicar ideias continuará a ser importante. Mas com a introdução de tecnologias no mundo jurídico o advogado do presente e do futuro terá de saber mais do que isso. Ele precisará interagir com sistemas inteligentes de automação, análise de conteúdo e gestão.

Nos Estados Unidos a tecnologia no Direito tem sido especialmente utilizada para auxiliar os advogados no gargalo jurídico, que consiste no *discovery*. Uma fase pré julgamento em que as partes, tanto no âmbito civil ou penal, por meio de seus advogados, podem solicitar a produção de determinadas informações, envolvendo divulgação de documentos, elaboração de questionários etc. Em outras palavras, a fase envolve a divulgação de informações que sejam necessárias para o julgamento. Nesse cenário a tecnologia auxilia na busca por informações importantes, bem como o armazenamento de tais informações, que geralmente envolvem um alto volume de informações.

No Brasil a realidade é diferente. De acordo com dados divulgados pelo CNJ [3] o ano 2016 finalizou com 79,7 milhões de processos em tramitação. Verifica-se um aumento de 29,4 milhões se comparados ao ano de 2015, refletindo um aumento de 5,6% e a tendência é aumentar a cada ano. O Poder Judiciário possui um grande volume de processos e é nesse cenário que se encontram ferramentas tecnológicas para auxiliar na automação e gestão de documentos jurídicos, monitoramento e extração de dados públicos, resolução de conflitos online, oferta de profissionais, dentre outros serviços jurídicos [4].

E acreditamos que um dos maiores diferenciais de um profissional será exatamente o de desenvolver a capacidade de programar lógica jurídica, criando o processo de automação de documentos jurídicos.

Automatizar a produção de um documento consiste em desenhar fluxos lógicos para que a partir de inputs esperados (questionário dinâmico e/ou algoritmos que interpretam texto), obtenha-se uma série de outputs desejados (teses, cláusulas e argumentos). É tornar conhecimento jurídico em *software*. Por meio de uma plataforma digital os usuários (que não necessariamente precisão ser pessoas que atuam no âmbito jurídico) irão acessar esse conhecimento automatizado para criar, editar, colaborar e até assinar documentos. Qualquer conteúdo que tenha padrões recorrentes pode ser automatizado (e por incrível que pareça todos possuem).

Assim, os profissionais poderão:

(a) **Criar documentos jurídicos em fração do tempo original**;

(b) **Reduzir custos**;

(c) **Melhorar qualidade e consistência** e

(d) **Extrair dados altamente estruturados para análise estatística**.

Venha nos acompanhar nessa jornada de novas descobertas e o desenvolvimento de novas habilidades,

preparando-se para as novas oportunidades que o Direito e tecnologias disruptivas estão proporcionando.

O que é Looplex e como ela se insere no mercado jurídico?

A Looplex é uma empresa de soluções de tecnologia para o mercado jurídico (legaltech ou lawtech) fundada por advogados, engenheiros e matemáticos voltada para automação de documentos jurídicos. A empresa desenvolveu um *software* que utiliza algoritmos de processamento de linguagem natural para gerar contratos, petições e outros documentos que hoje são feitos de forma artesanal.

A produção de documentos é simples e intuitiva. O usuário terá acesso a uma plataforma *online* que armazena seus documentos. Ao acessar o sistema, o usuário escolherá o *template* a ser produzido. Ao indicar o *template* a ser utilizado, a interface fornecerá uma entrevista guiada e com poucos cliques os documentos são construídos de forma dinâmica. Em linhas gerais, o *software* interpreta as respostas e, com base nelas, atualiza as perguntas seguintes.

Além disso, o software permite a utilização do próprio estilo do usuário (fonte, espaçamento, margens, identação, inserção de logo etc), bem como realizar a impressão e exportação para o formato que desejar (PDF, Word (docx) ou em formatos específicos para integração com outros sistemas).

Em suma, a Looplex ao promover a automação de documentos jurídicos, objetiva: (a) dar agilidade na produção de documentos, promovendo uma redução significativa no tempo de elaboração de documentos jurídicos, (b) realizar a produção de estatísticas a partir dos documentos produzidos, (c) implementar a padronização e dar consistência aos documentos para escritórios, empresas, instituições públicas e privadas etc[1].

A atuação na Looplex nos permitiu identificar uma lacuna de profissionais no mercado jurídico que possam lidar com as demandas decorrentes dos desafios relacionados às tecnologias, especialmente no âmbito da automação de documentos jurídicos. Deste modo, esse livro é uma ótima oportunidade para compartilhar nossos conhecimentos adquiridos na prática, estimulando outros profissionais a se capacitarem e desenvolverem as habilidades necessárias para atuar no processo de automação de documentos jurídicos.

Afinal, quais as profissões decorrentes da automação?

No decorrer do processo de automação identificamos a existência de três profissionais que surgem nesse cenário, a figura do **engenheiro jurídico**, **arquiteto jurídico** e do **analista jurídico**. Não obstante, com a evolução e novas demandas surgindo no mercado jurídico, certamente novas profissões serão criadas.

Montagem de árvores: a figura do arquiteto

O arquiteto jurídico possui uma visão estratégica do projeto, lidando com os dados jurídicos estruturados, gestão do conhecimento, procedimentos e pessoas usando ferramentas metodológicas para estruturar os projetos de automação. A função principal do arquiteto jurídico é especificar e delimitar os objetivos de cada projeto, organizando o conhecimento jurídico e possuindo o papel de gerente das equipes que irão conduzir os projetos.

[1] Para mais informações, sugerimos o acesso ao site da Looplex `http://www.looplex.com.br`

Codificador: a figura do engenheiro

O engenheiro jurídico atua na estruturação de conhecimento de dados e informações, buscando identificar os passos e critérios usados para atender as possíveis permutações decorrentes de um mesmo caso. Durante o processo de estruturação do conhecimento jurídico sua atuação também envolve a tradução de tais conhecimentos em linguagem de máquinas, ou seja, interpretar tais informações por meio de codificação. Em poucas palavras, o engenheiro jurídico possui a função de executor de especificações e projetos.

***Business Intelligence*: a figura do analista**

O analista jurídico é o profissional encarregado de estruturar e organizar dados e informações decorrentes do processo de automação, oferecendo o devido suporte à gestão dos negócios do escritório ou empresa. Esse profissional possui condições de monitorar e extrair dados e estatísticas a partir dos documentos gerados pelo *template*, como por exemplo, identificar o número de documento produzidos no mês, quantos documentos foram produzidos por cada usuários do sistema, ou seja, a taxa de produtividade de cada usuário do sistema, a taxa de ganho dos documentos produzidos perante o Poder Judiciário, dentre outras informações. Tais informações auxiliam na definição de estratégias relacionadas ao negócio, possuindo um papel central em como os serviços jurídicos são e poderão ser prestados.

É importante mencionar que mesmo tendo funções e propósitos distintos, esses dois primeiros profissionais atuam conjuntamente e devem ser capazes de compreender, abstrair, interpretar, analisar e modelar conceitos jurídicos. Assim, é muito importante que tenham domínio de gestão de conhecimento de dados e informações, sendo responsáveis pela: (a) coleta e organização de dados relevantes para o processo de automação, (b) estruturação da árvore de decisão que fundamenta os documentos, (c) utilização de técnicas de composição textual para melhor atender as especificidades jurídicas, (d) aplicação de boas práticas de especificação de perguntas e informações complementares que deverão ser disponibilizados aos usuários finais do documento jurídico, atuando conjuntamente para o desenvolvimento de projetos.

Para entender melhor as atividades que envolvem a atuação de arquitetos e engenheiros jurídicos dê uma olhada nos subitens 2.5 Engenharia Jurídica e 2.6 Arquitetura Jurídica nesse livro.

Por que ser um arquiteto ou engenheiro jurídico?

Até esse momento você conseguiu compreender o cenário em que o Direito e a tecnologia se encontram e como a automação de documentos jurídicos é uma realidade. Perfeito! Mas quais motivos o levariam a se aventurar em um novo conhecimento se a sua atuação vai "muito bem, obrigado"?

Caso ainda tenha dúvidas, temos alguns motivos pelos quais acreditamos ser importante ter os conhecimentos e habilidades de um arquiteto ou engenheiro jurídico:

(a) **Valorização no mercado profissional**.
O que diferencia um profissional de outro certamente não são as habilidades que possuem em comum, mas sim, as características e habilidades diferentes. Há um tempo atrás o diferencial de um profissional no mercado de trabalho, independentemente da área de atuação, era o domínio da língua inglesa. Atualmente dominar o inglês não é um diferencial e sim um requisito mínimo para que os profissionais tenham boas oportunidades.

Em 2013, famosos como Bill Gates, Mark Zuckerberg, o cantor will.i.am, dentre outras celebridades, se uniram para promover uma campanha voltada para a importância de aprender programação. No vídeo[2], compartilharam suas experiências e ressaltaram a importância de aprender a linguagem de programação.

Saber programar com alguma linguagem por si só não é uma vantagem, mas sim, o aprendizado decorrente do processo de estruturação do raciocínio utilizado para resolver problemas. É justamente esse processo de organização e estruturação de conhecimento, especialmente jurídico, é que enfatizamos no curso. Essas habilidades não são exigidas apenas na área do Direito, como também em engenharia, medicina, agricultura etc, sendo que os profissionais que desenvolverem tais habilidades certamente terão maior destaque no mercado profissional e, consequentemente, acesso à melhores oportunidades de trabalho.

(b) **Foco na prestação de serviços jurídicos**.
O dia a dia da prestação de serviços advocatícios não é tão glamouroso como muitos imaginam. A rotina é repleta de atividades burocráticas que muitas vezes tomam mais tempo do que a própria resolução de problemas jurídicos. Perdem-se horas (muitas vezes não cobradas dos clientes) no protocolo de processos, a revisão minuciosa de concordância de gênero (masculino e feminino) e número (singular e plural) em petições e contratos, dentre outras atividades.

Nesse sentido, a observação de Mary Bonsor [5] é muito interessante:

> 'Tech will be very helpful in creating opportunities for lawyers to focus on interpreting law, doing their legal role rather than an administrative side of things that currently occupies a lot of time(...)'. 'Client relationships and interpretation of law will be hard for machines to master, even if they can help humans in doing so. The arrival of tech will certainly promote legal analysis, but it will not replace much of law as we know it.'

A ideia é que o arquiteto e o engenheiro jurídico coexistam em sintonia, de modo a distribuir tarefas tais como a estruturação, organização e codificação do conhecimento jurídico para a automação de documentos, bem como a realização de pesquisas, elaboração de novas teses e cláusulas. Nesse sentido, o profissional do Direito tem a oportunidade de dedicar grande parte de seu tempo em questões jurídicas. Em suma, ter tempo para aprimorar seu conhecimento técnico no âmbito do Direito, agregando valor às atividades importantes para seus clientes.

(c) **Visão holística de questões jurídicas**
A prestação de serviços jurídicos tradicionais envolve de um lado o cliente compartilhando seus problemas e de outro o operador do Direito tentando aplicar seus conhecimentos jurídicos técnicos ao caso prático para sugerir soluções aos problemas apresentados. Nessa forma de prestação de serviços jurídicos o que agrega valor é a informação fornecida pelo operador do Direito.

Por exemplo, em uma contestação trabalhista a função do operador do Direito é redigir uma peça que contemple todos os pedidos indicados na inicial, bem como a inclusão de pedidos adicionais, caso aplicável, sendo todos esses temas referentes ao caso concreto indicado na inicial.

No processo de automação de documentos jurídicos a dinâmica de prestação de serviços é diferente. O

[2] `https://www.youtube.com/watch?v=nKIu9yen5nc`

valor da prestação de serviços não advém da resposta fornecida aos casos concretos, mas sim, ao uso do sistema que acaba gerando as informações. Portanto, a prestação de serviços jurídicos fornecida por um arquiteto jurídico é a estruturação e organização de informações, assim como o engenheiro jurídico é responsável pela interpretação de linguagem natural para a linguagem máquina.

No mesmo exemplo utilizado, a função do arquiteto e do engenheiro jurídico é mapear e identificar todos os pedidos que podem ser solicitados pelas possíveis partes reclamantes, bem como todos os pedidos adicionais que poderiam ser solicitados pelas reclamadas, ou seja, é necessário ter uma visão mais ampla e aprofundada do Direito. Exigi-se, portanto, um conhecimento jurídico aprofundado na área trabalhista.

Trata-se de uma nova forma de lidar com o Direito, proporcionando ao operador do Direito um conhecimento amplo e completo acerca de variados temas, estimulando a pesquisa e aprofundamento no conhecimento jurídico.

(d) **Interdisciplinaridade constante com outras áreas**.
Há décadas o ensino do Direito em grande parte das instituições estimula a compartimentalização de disciplinas, reproduzindo especialistas em áreas específicas. Notam-se poucas instituições que promovem a formação de operadores do Direito com conhecimento em outras áreas ou ao menos que possam dialogar com diferentes áreas.

Na atuação prática do arquiteto e do engenheiro jurídico a necessidade de interação com outras áreas é muito importante. Utilizamos os conhecimentos e ferramentas usualmente utilizados em outras áreas. Para realizar a especificação de projetos, é muito importante identificar os possíveis problemas e pensar em alternativas em como resolver tais problemas, assim utilizamos o método do *Design Thinking* para nos auxiliar nesse processo, sendo esse processo amplamente utilizado por *designers*.

Já para organizar a produção de *templates* (nossos documentos produzidos) e proporcionar uma gestão de projeto mais produtiva e eficiente, são utilizados **métodos ágeis**, que basicamente consistem em um conjunto de práticas voltadas para atender as demandas dos clientes adicionando valor ao seu produto, promovendo uma boa interação entre os membros da equipe que desenvolvem o produto, aplicando sempre uma excelência técnica, entre outros valores. Os métodos ágeis são amplamente utilizados para desenvolvimento de *softwares*. Para mais informações sobre métodos ágeis verifique o item 3.2. do presente livro.

Em se tratando de programação, conhecimentos básicos de teoria dos conjuntos e lógica formal, são muito importantes para auxiliar na estruturação de frases e transformando as informações em dados interpretados pelas máquinas.

Enfim, a atividade tecnológica jurídica é permeada por conhecimentos e ferramentas utilizadas em outras áreas, proporcionando um contínuo estímulo ao aprendizado e o conhecimento de outras áreas. Tenha em mente, você sempre irá aprender coisas novas!

Esses são os principais conhecimentos e habilidades que visualizamos na atuação do arquiteto e do engenheiro jurídico, bem como a prática de novas habilidades e quais conhecimento serão desenvolvidos. Se mesmo assim, você ainda não acredita na necessidade de ter tais conhecimentos, então é o momento de aplicar o "tratamento de realidade" e compartilhar alguns dados e informações a respeito.

Em uma pesquisa recente, a empresa Thomson Reuters utilizando como base de sua pesquisa informações divulgadas pela Organização Mundial de Propriedade Intelectual, informa que nos últimos 5 (cinco) anos subiu em 484% o número de requerimento para registro de patentes relacionadas aos novos serviços jurídicos que envolvem tecnologia no mundo inteiro[3].

Esses números revelam a expansão de novos modelos de serviços jurídicos que despontam no mercado, sendo grande parte dos modelos relacionados à tecnologia, demonstrando uma verdadeira alteração na forma como os serviços jurídicos são prestados. Portanto, não se trata de uma tendência do futuro, mas sim do presente.

Ao estudar os impactos da tecnologia na carreira jurídica, o professor Richard Susskind [6], propõe as possibilidades de novos serviços jurídicos que poderão ser prestados, como:

- ***The Legal knowledge engineer***. Em se tratando de sistemas automatizados que objetivam padronização, os profissionais serão requeridos a organizar e estruturar informações e processos jurídicos em *softwares*. *Legal knowledge engineer* é o termo utilizado, que nos inspirou para designar as competências e habilidades do "engenheiro jurídico".

- ***The Legal tecnhnologist***. Serão necessários profissionais que compreendam e tenham o domínio de problemas jurídicos e ao mesmo tempo atuem na área de engenharia de sistemas e/ou gestão de tecnologias, para que desenvolvam sistemas inteligentes que resolvam problemas jurídicos, enfim, aqueles profissionais que projetarão os sistemas.

- ***The Legal hybrid***. A forma como os serviços jurídicos serão prestados indica uma alteração na forma de resolver problemas, passando de resoluções particulares (resolução de casos individuais e específicos) para estruturar e organizar informações com o intuito de resolver vários problemas jurídicos. Desta forma, os profissionais deverão agregar valor não apenas com o conhecimento jurídico, mas também oferecer outros serviços que possam agregar valor aos seus clientes. Por exemplo, desenvolver serviços de estratégia de mercado com base no nicho de clientes que os respectivos escritórios e empresas possuem ou pretendem alcançar, serviços de consultoria para otimizar a escalabilidade do negócio ou explorar novos serviços a serem prestados.

- ***The Legal Process Analyst***. Como o próprio nome explica, é o profissional responsável por analisar o serviços jurídicos a serem prestados e discretizá-los, fracionando em porções menores estabelecendo como deverão ser executados, bem como desenvolvendo maneiras mais eficientes em executar tais tarefas. É um profissional voltado à gerencia das atividades jurídicas a serem desenvolvidas.

- ***Legal Project Manager***. Após a atuação do *legal process analyst*, em organizar e estruturar as especificações de cada tarefa, o *legal project manager* deverá colocar em prática tais especificações. Em outra palavras, deverá assegurar que as atividades serão repassadas aos profissionais que se mostram mais adequados em executar tais tarefas, bem como garantir que as tarefas serão executadas. Assim, como o analista de processos jurídicos, também é um profissional voltado à gerencia das atividades jurídicas a serem desenvolvidas.

- ***The Legal Data Scientist***. Esse profissional é aquele que deverá ter o domínio de ferramentas e técnicas que objetivem manipular e analisar uma quantidade significativa de informações. Para esse profissional serão necessários conhecimentos em matemática, programação, ou seja, exige-se uma atuação interdisciplinar.

[3]Os dados completos dessa pesquisa estão no site `https://www.thomsonreuters.com/en/press-releases/2017/august/thomson-reuters-analysis-reveals-484-percent-increase-in-new-legal-services-patents-globally.html`

- ***The Research and Development Worker***. A atuação deste profissional consiste em realizar pesquisa e desenvolvimento de novos produtos e soluções jurídicos. A pesquisa e o desenvolvimento possibilitarão a descoberta de novas capacidades, produtos, técnicas e serviços que serão prestados e demandados no mercado jurídico, assim como já está acontecendo com o mercado de *lawtechs* ou *legaltechs*.

- ***The ODR Practioner***. As resoluções de disputa *online* estão ganhando espaço no mercado como um todo e poderão ganhar espaço no mercado jurídico, um novo profissional deverá ter condições de conduzir tais acordos realizados de forma *online*. Deste modo, novas habilidades e capacidades serão exigidas desse profissional.

- ***The Legal Management Consultant***. No Direito a parte de gestão normalmente é reservada aos sócios da empresa ou escritórios, que devem se preocupar com a contratação de funcionários, valor de infraestrutura e redução de custos, dentre outras atividades de um administrador. No novo cenário em que Direito e tecnologia estão diretamente interligados, esse pode ser um nicho de prestação de serviços e não apenas uma parte necessária na atividade de escritório. Tais profissionais serão os denominados gerentes consultivos jurídicos.

- ***The Legal Risk Manager***. Existem poucas iniciativas no âmbito jurídico com o propósito de mitigar riscos na prestação de serviços jurídicos. Em grande parte, os escritórios de advocacia e empresas, optam por solucionar problemas pontuais que acabam aparecendo no dia a dia. O gerente de riscos legais, possui a função de identificar, monitorar e controlar problemas, desenvolvendo metodologias, regras, técnicas e processos com a finalidade de mitigar os riscos inerentes a atividade de prestação de serviços jurídicos.

Essas novas profissões identificadas pelo professor Richard Susskind demonstram as novas habilidades que serão exigidas dos profissionais do Direito, bem como as possibilidades de atuação que estão surgindo no mercado.

1.2 Organização do livro

Com o intuito de auxiliar os profissionais do Direito a compreenderem os principais aspectos que envolvem o processo de automação e engenharia jurídica, este livro está estruturado em sete partes. O primeiro capítulo é a introdução, reservada para explicar as noções básicas que envolvem a tecnologia e Direito, dando subsídios para que o leitor possa compreender o cenário em que o curso está inserido.

Já o segundo capítulo traz as principais definições que serão utilizadas nesse livro, auxiliando na compreensão de termos que não são familiares aos profissionais do Direito.

O terceiro capítulo descreve os fundamentos de lógica e programação necessários para que os alunos tenham a base e possam realizar a estruturação lógica de documentos. Deste modo, serão abordados conceitos básicos de Teoria dos Conjuntos, Lógica Proposicional, Fundamentação Algorítmica e alguns temas considerados como paradigmas clássicos de Programação.

O quarto capítulo se propõe a esclarecer as principais atividades desenvolvidas, metodologias e ferramentas utilizadas pelos profissionais que realizam a arquitetura jurídica. Serão abordados temas como a ferramenta metodológica de *design thinking* e sua utilização para profissionais que atuam com o Direito, as principais

metodologias ágeis para auxiliar no planejamento, organização e desenvolvimento de produtos, as principais ferramentas de análise utilizadas pelos arquitetos jurídicos, estruturação e organização da análise de requisitos solicitados por clientes, estruturação de documentos jurídicos e elaboração de *templates*.

Nesse mesmo sentido, o quinto capítulo se propõe a elucidar sobre as principais atividades realizadas, metodologias e ferramentas utilizadas tanto pelos arquitetos quanto engenheiros jurídicos.

O sexto capítulo propõe a retomada dos principais conceitos de programação com o intuito de introduzir a programação com a linguagem Lawtex. Nesse contexto será introduzida a sintaxe da linguagem Lawtex, a topologia dos templates em Lawtex, os operandos e suas propriedades, as classes de operandos, as declarações e tipos e algumas operações básicas todas referentes à linguagem Lawtex.

Por fim, a conclusão apresentará os principais pontos abordados no presente livro, fazendo uma relação entre programação e engenharia jurídica.

2 Definições Gerais de Direito Computacional

Para uma eficaz assimilação do conhecimento por parte do leitor, vamos definir alguns conceitos básicos do direito computacional, conforme segue.

2.1 Direito Computacional

Para os fins desse trabalho, define-se como *legal informatics* toda aplicação de informática ao Direito, seja para pesquisa de precedentes, leitura de documentos, produção de estatística, entre outras.

O campo compreende, portanto, *software* de pesquisa, *software* de leitura de documentos, *software* de simulações de balística, comparação de assinaturas e textos e quaisquer outros instrumentos de perícia, *software* de identificação de fala e transcrição de grampos etc. Por outro lado, define-se Direito Computacional como um dos campos da informática jurídica, sendo o campo que estuda formas de expressão de lógica jurídica em linguagem computacional.

Como a redação de documentos depende do exercício de raciocínios jurídicos, as ferramentas de *document assembly*, geradoras automáticas de texto, fazem parte desse campo.

2.2 Tecnologias de análise de documentos e de disputas

O objetivo desse trabalho não envolve o aprofundamento em novas tecnologias utilizadas no cenário nacional e mundial para análise de documentos e disputas, mas se detém a automação de documentos jurídicos. Não obstante, para aqueles curiosos sobre o tema, apresentamos alguns conceitos que surgem no âmbito de novas tecnologias e indicação de leitura para aprofundamento.

Early Case Assessment (ECA)

A doutrina oferece várias definições, mas basicamente trata-se de uma metodologia de gerenciamento (ou juridicamente categorizada como uma medida alternativa de resolução de disputa) utilizada preventivamente em processos jurídicos no âmbito contencioso, utilizando-se de diversas ferramentas para verificar fatos, legislação ou quaisquer outras informações relevantes que possam auxiliar na compreensão e avaliação das minúcias de um determinado processo, no desenvolvimento de estratégias e elaboração de planos de acordo, caso aplicável, otimizando o tempo, risco de perda e principalmente os custos decorrentes de um processo contencioso.

Para conhecer melhor a metodologia ECA, recomendamos a leitura de um guia prático desenvolvido pela

Symantec©, denominado "*Understanding the Impact of Early Case Assessment to Your Organization*" que está disponível na Internet [4].

Para nós, interessam as ferramentas tecnológicas que podem auxiliar a ECA, como *predictive coding* a seguir definidos.

Predictive Coding, Technology Assisted Review (TAR) ou *Computer-Assisted Review*

Aplicando o conceito no campo jurídico, *predictive coding* consiste na utilização de algoritmos que se baseiam em exemplos inseridos por humanos, para replicar a lógica inserida, categorizando e revisando um número considerável de documentos, sem a necessidade de uma revisão de humanos. A utilização de *predictive coding* alterou o processo de análise de documentos, especialmente no Direito Norte Americano que possui uma etapa pré-processual denominada *Discovery*, análise previamente realizada por advogados. Na prática, é necessário que um advogado que tenha experiência insira todos os parâmetros de pesquisa a serem considerados para a extração de informações relevantes. Assim os algoritmos captam um padrão de parâmetros e passam a adotá-lo na análise de informações, ou seja, um trabalho que exige a interação homem-máquina. Para entender os pontos principais argumentos levantados na utilização do *predictive coding* recomendamos a leitura do artigo *Predictive Coding* [7].

2.3 *Smart Documents*

A expressão de raciocínios jurídicos em linguagem computacional permite a criação do que chamamos de *smart documents*. Trata-se de documentos que, por serem expressos também em linguagem máquina, sejam automaticamente implementados no mundo real. O poder de um documento assim é tão maior quanto mais integradas forem as plataformas e serviços. No limite podemos pensar em um contrato que, após celebrado, envia e-mails notificando as partes, faz transferências de propriedade em cartórios, transfere recursos de contas bancárias, calcula valor de obrigações e multas periodicamente etc. São documentos que fazem a própria gestão de seu ciclo de vida.

Com um documento já expresso em linguagem humana e em linguagem computacional, ou seja, com contratos estruturados, a produção de dados sobre esses documentos e integração com novas funcionalidades e plataformas é praticamente automática. Nesse ponto a engenharia jurídica não só muda a forma como os operadores do Direito costumam trabalhar, mas ela fomenta novas atividades, como a de estruturar não só o que os documentos jurídicos prescrevem, mas também a sua implementação.

2.4 Automação e Interpretação

Não é objetivo desse trabalho fazer uma reflexão teórica profunda sobre a relação entre engenharia jurídica e teoria do Direito. Isso será feito em publicações futuras. Nessa introdução nos parece satisfatório apresentar, de maneira resumida, nossa resposta sobre duas inquietações comuns: o que acontece com a interpretação do Direito? Os operadores do Direito passam a ser desnecessários?

A automação, de fato, surpreende. Ela parece ser uma ruptura brusca na forma de se pensar o Direito, mas ela não é o início de um processo. Ela é resultado dele. Nesse ponto uma analogia com outras indústrias parece ser pertinente. Imagine três estágios de produção de móveis de madeira: (1) a produção por um

[4] Espero que o link `http://cyfor.co.uk/files/2012/07/Understanding-the-Impact-of-Early-Case-Assessment.pdf` ainda funcione :-)

artesão, que centraliza em si todas as etapas de produção do móvel, esculpindo a madeira, pregando as peças e pintando o produto final; (2) a manufatura, em que todas essas atividades são divididas entre grupos especializados, cada um fazendo apenas uma das etapas de produção e (3) a produção industrial em que cada uma das etapas é feita por máquinas, com poucos humanos supervisionando a produção e mantendo as máquinas operacionais.

Diante disso um artesão clássico olha para a automação da produção e lamenta: "a arte da produção de móveis morreu com a automação!". Se interpretarmos arte nesse contexto como o valor que atribuímos a atividades e habilidades humanas, de fato a arte morreu. O valor intrínseco da capacidade humana de criar um móvel do começo ao fim foi se esvaziando, o que ainda não aconteceu, por exemplo, com a arte de tocar instrumentos, que ainda é valorizada embora máquinas e computadores possam produzir o mesmo som com qualidade equivalente. Contudo, a observação do artesão está errada, pois a arte não morreu com a automação, mas sim com a massificação promovida no estágio (2).

Veja, já na manufatura massificada, não havia necessariamente valor intrínseco ou artístico na atividade de fazer um móvel. Tanto é verdade que talvez nenhum dos trabalhadores da linha de produção não soubesse fazer um móvel inteiro, do começo ao fim. A "morte da arte" ocorre no momento da massificação da produção, a automação, assim como a linha de produção, é apenas mais uma das formas de massificação. Com o Direito o mesmo é verdadeiro.

A automação não é a morte da interpretação. A massificação é. A maior parte dos escritórios reproduz modelos e técnicas de maneira acrítica no maior volume possível. Nada diferente do que um computador faria. A atividade interpretativa já é restrita a poucos operadores do Direito, aqueles que ocupam posições de prestígio. Isso não é necessariamente ruim. A massificação da produção de móveis permitiu um aumento da produção e redução do preço. O acesso da população aos móveis é maior hoje do que foi na idade média. Isso vale também para o Direito ou para qualquer outro mercado.

A massificação, independentemente do contexto em que se insere, é inclusiva. Hoje o acesso à justiça é maior do que no passado, em que mais da metade da população sequer podia votar. De qualquer forma, alguém terá de alimentar os sistemas. O Direito é uma construção humana e está em constante transformação. A interpretação criativa existirá, mas ficará restrita aos que programam o sistema, ao passo que aqueles que consomem o sistema apenas seguirão os procedimentos definidos, numa interpretação conversacional do que o sistema exige. Logo, a interpretação continuará existindo, mas seu espaço será restrito.

A segunda pergunta tem grande relação com a primeira. Os operadores não são desnecessários, mas algumas atividades que eles costumavam executar sim. Antes das ferramentas eletrônicas de busca de jurisprudência fazia parte da atividade do operador do Direito consultar revistas de tribunais para selecionar precedentes. Hoje essa atividade não existe mais, a busca não envolve o deslocamento até uma biblioteca e a leitura de uma revista. Talvez com a automação a reprodução massificada de teses e raciocínios (desde aqueles relativos a um pedido de horas extras até aqueles relativos às *representations* e *warranties* de um *share purchase agreement*, indistintos do ponto de vista de complexidade lógica e computacional) empregue menos pessoas, mas o acesso à justiça seja maior, mais preciso e mais uniforme. De qualquer forma, os operadores do Direito não vão deixar de existir, mas as competências e habilidades que se espera deles vai mudar profundamente.

2.5 Engenharia Jurídica

Adotamos um conceito de engenharia jurídica inspirado em Susskind [8]. Nesse sentido a engenharia jurídica é um sinônimo de engenharia da informação jurídica. No paradigma atual de prestação de serviços jurídicos os operadores do Direito são remunerados em virtude da assimetria informacional que eles têm em relação ao cliente. O cliente paga para obter uma orientação ou para ver seu problema solucionado. Dessa forma, em cada caso o operador do Direito faz um raciocínio jurídico, relacionando as informações que tem com o caso concreto para oferecer a informação que o cliente deseja. Porém, com a informatização a produção da informação é massificada.

Oferecer a informação adequada ao caso concreto perde muito valor, de forma que a atividade do operador do Direito passa a ser a organização de informação. O cliente deixa de pagar pela resposta, mas paga pelo uso de um sistema gerador de respostas, que foi previamente programado por operadores do Direito. Essa atividade é mais abstrata que a atividade atual (realizada de forma artesanal). No lugar de raciocinar juridicamente para resolver um caso, o que se busca é identificar, universalmente, os passos e critérios usados para resolver todas as permutações de uma mesma categoria de caso. Trata-se de estruturar o conhecimento dos operadores do Direito. E a engenharia jurídica é justamente essa organização. No lugar de se pensar, por exemplo, "cabe uma indenização nesse caso?", mas sim "quais passos e raciocínios são necessários para determinar se uma indenização é cabível em um caso?".

Uma pergunta que surge, nesse momento, é "qual é a diferença, então, entre a engenharia jurídica e a doutrina ou o esforço de codificação do século XIX". No limite, nenhuma. Tanto a codificação quanto a elaboração de doutrinas segmentadas partilham com a engenharia jurídica o desejo de sistematização. Num olhar mais específico, contudo, a engenharia jurídica é uma "abstração pragmática". Ela se aproxima da doutrina ou da codificação no desejo de generalidade, mas se aproxima da prática jurídica no seu desejo de aplicabilidade. Em geral, exercícios de abstração no Direito buscam unificar suas interpretações em torno de um sistema coerente de princípios, conceitos e valores. Aliás, se o legislador utiliza, por exemplo, no Direito Tributário um conceito de contrato diferente do utilizado no Código Civil ou se o Código de Processo Civil utiliza em artigos distintos a palavra "recurso", a comunidade jurídica reage com críticas e é feito um esforço para manter a coerência do sistema. Por isso é comum ler na doutrina que o legislador cometeu um "erro". Assim, dificilmente uma obra doutrinária é meramente descritiva. Ela mescla descrição do passado com uma interpretação criativa que dê conta de tal passado de maneira coerente (Dworkin [9] provê uma discussão mais profunda sobre o tema).

A engenharia jurídica se afasta, portanto, da doutrina, porque ela é mais rasa, no sentido de que quer apenas sistematizar os passos para solução de um ou mais problemas. Sua preocupação com a coerência do sistema é secundária, e só surge se isso impõe um obstáculo para resolver o problema a ser tratado. Do contrário, basta organizar uma árvore de decisão e, em seguida, uma expressão computacional daquele processo decisório. Aliás, essa é outra diferença fundamental entre a doutrina e a engenharia jurídica. A linguagem por meio da qual a engenharia jurídica expressa sua sistematização é a computacional.

2.6 Arquitetura Jurídica

A figura do diretor de conhecimento (do inglês, *Chief Knowledge Officer - CKO*) tem se destacado nas últimas décadas dentro do cenário jurídico. Sua função é gerir o capital intelectual da empresa, ser responsável por reunir todo o conhecimento da organização e consolidá-la de modo estruturado. A mera organização de minutários e manuais de procedimentos já não é uma atribuição suficiente para que um profissional desse ramo se estabeleça em sua plenitude. O uso de ferramentas de gestão e suporte a produção de conteúdo

é indispensável. O CKO do futuro deve imergir no mundo tecnológico, com plenos domínios tanto de ferramentas de gestão de processos que orbita em torno de *legal tech*, quanto de gestão de pessoas.

No âmbito de automação de documentos, se demonstra que o controle do processo de composição e manutenção efetiva de documentos jurídicos por meio de ferramentas computacionais é algo de grande valor para os profissionais do Direito. Para um CKO ou mesmo para os coordenadores de projeto, esse tipo de ferramenta é de especial interesse, não só para a gestão de processos e pessoas, mas como um agente de implantação do conhecimento. O gestor assume uma posição não mais como expectador passivo, mas como agente ativo no processo. Suas atribuições variam entre definição de esquemas para elaboração das teses, restrição de caminhos lógicos na argumentação, revisão ortográfica, revisão de formatação do texto, análise técnica do conteúdo jurídico, e outros aspectos. A notícia boa é que os tempos da "marmota"[5] estão prestes a se findar: produção de documentos e gestão de relatórios que demandariam um tempo inestimável, especialmente para documentos produzidos em massa, hoje estão a um clique de obter velocidade, qualidade e consistência.

O arquiteto jurídico surge como um profissional que lida com a gestão do conhecimento, processos e pessoas usando ferramentas e procedimentos formais para definição de projetos de automação e organização do *know-how* de uma corporação. Dentre as ferramentas formais, o arquiteto jurídico deve ter conhecimentos profundos de temas concretos do direito, dominar ferramentas de projeto, tais como árvores de decisão [10], linguagens de marcação legal [11], métodos ágeis [12] para gestão de pessoas, bases de *design thinking* [13], bem como outras ferramentas para gestão informacional (mapas mentais [14], diagramas entidade relacionamento [15], etc).

Embora as tarefas do arquiteto jurídico não se fundamentam em codificação e elaboração de algoritmos, seu nível de especificação deve ser técnica e precisa, conscientes do que pode ser feito ou não com os recursos que se tem. Num exemplo de mercado financeiro, não se pode programar um robô que opere na bolsa de valores com comandos do tipo: "Compre na baixa e venda na alta". Para um programador júnior, uma instrução como essa soa ridícula e infundada.

O papel do arquiteto não é fazer uma lista de pedidos e torcer para que tudo seja cumprido e executado pelos engenheiros. Sua função requer que se conheça o básico de programação e que domine a linguagem lógica e conceitos de representação do conhecimento. Quanto à máquina aprender o que queremos... Quem sabe num futuro próximo, a própria máquina nos retribua pelas informações de alto nível técnico, estruturadas, confiáveis e de alta qualidade que a ela submetemos.

Quanto às responsabilidades dos arquitetos e engenheiros dentro de um projeto de automação jurídica, existem pontos a serem considerados. A Figura 2.1 mostra um gráfico de atribuições dominantes em atividades de engenharia e arquitetura jurídica. Note que as tarefas mais concretas, ou seja, de mais baixo nível abstrato são atribuídas a engenheiros jurídicos, tais como codificação, testes e detalhamento da execução do projeto. Já as tarefas mais complexas, que exigem um nível maior de abstração, conhecimento macro do tema e gerenciamento de equipe, são mais vinculadas a arquitetos. Vale salientar, que o compromisso do arquiteto jurídico é especificar o projeto, enquanto que o engenheiro jurídico precisa compreender o projeto (com todo seu formalismo inerente) a fim de implementá-lo. A rigor, as especificações técnicas do projeto devem ser suficientes para que um engenheiro jurídico o codifique.

Segundo palavras de um conceituado arquiteto Prof. Dr. Antonio Cláudio Pinto Fonseca, "A arquitetura é um exercício intelectual, enquanto que a engenharia é um exercício prático". Em quase todos os casos,

[5] Referência dada ao filme *Feitiço do Tempo* em que o personagem interpretado por Bill Murray revive o mesmo dia inúmeras vezes sem fim

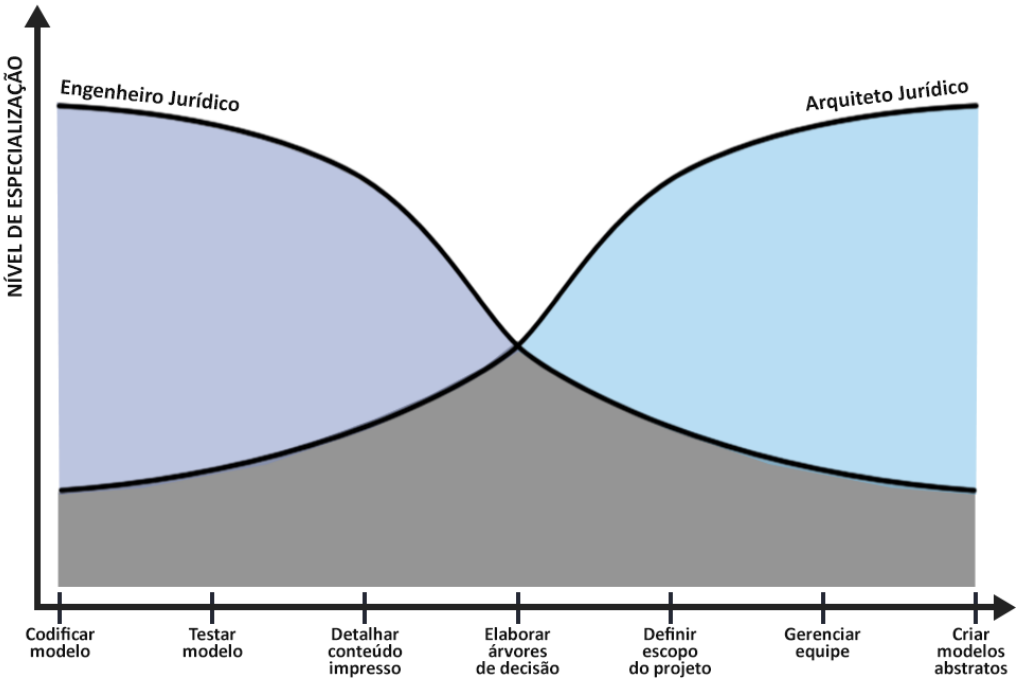

Figura 2.1: Gráfico de responsabilidades de engenheiros e arquitetos jurídicos.

arquitetos se especializam em projetos e composição criativa e os engenheiros lidam com detalhes técnicos de implementação e execução efetiva do projeto. Um paralelo interessante entre arquitetos e engenheiros jurídicos ocorre no ofício da construção civil. Nesse caso, o arquiteto organiza e projeta espaços segundo a funcionalidade, o conforto e a estética. Também coordena a construção (ou reforma) de edifícios, casas e imóveis em geral. Enfim, o arquiteto pensa em tudo: planta do imóvel, materiais usados na obra, usabilidade do imóvel, disposição dos móveis, ventilação e iluminação etc. Já o engenheiro civil é responsável por projetar, gerenciar e executar obras, bem como acompanhar a execução de construções e reformas de casas, prédios, viadutos, estradas etc. Ele também acompanha todas as etapas da obra, desde a análise do terreno, definição dos tipos de fundação, especificação das redes de instalação elétrica e hidráulica do edifício, além da supervisão de custos, segurança, prazos e qualidade. Nesse paralelo, observamos que o engenheiro é o agente executor, que acompanha todo projeto em sua minúcia técnica, enquanto que o arquiteto é o agente criativo, que elabora o projeto com os olhos de quem usufrui do próprio produto concebido.

É de se esperar que ambos os profissionais conheçam todas as ferramentas incluídos no ciclo de vida do projeto. Embora a programação (ou codificação) não seja o ponto forte de um arquiteto jurídico, ele deverá, ao menos, conhecer as possibilidades e limitações técnicas para codificar um procedimento jurídico automatizável, ou seja, deverá dominar elementos fundamentais da composição algorítmica e expressá-las por meio de um protocolo formal com alto-nível de abstração.

Obviamente, o intuito não é "escrever na pedra" quais as atribuições de cada profissional, a ideia central é traçarmos um perfil básico de cada profissional de acordo com as tarefas comuns no âmbito da automação jurídica. A evolução das profissões reposicionará por si só as atribuições de cada profissional, criando novas categorias, novos profissionais, novas funções, de acordo com os perfis predominantes de cada projetista.

3 Ferramentas para Automação Jurídica

Antes de entrarmos a fundo na programação e outros aspectos de codificação, vamos abordar algumas ferramentas que podem compor a caixa de ferramentas útil tanto para um arquiteto quanto para um engenheiro jurídico.

3.1 Bases do *Design Thinking*

O *design thinking* nada mais é do que uma abordagem metodológica que utiliza ferramentas, métodos e criatividade para solucionar problemas. Inicialmente concebida como uma solução voltada ao *design*, foi amplamente disseminada para outras áreas por sua abordagem criativa e colaborativa para a resolução de problemas.

Charles Burnette, reconhecido no âmbito do *design* define o *design thinking* como:

> "Um processo de pensamento crítico que permite organizar informações e ideias, tomar decisões, aprimorar situações e adquirir conhecimento."

Essa abordagem pode ser sintetizada em cinco passos principais para a resolução de problemas, conforme indicado na figura abaixo:

Imersão

Essa etapa é voltada para compreender a extensão do problema a ser resolvido. Deste modo, trata-se de uma etapa exploratória em que é estimulada a compreensão dos diversos atores, situações, posicionamentos, sendo realizado um verdadeiro mapeamento acerca de todas as questões que se aproximam do contexto do problema. Também conhecida como **etapa da Empatia**, momento em que os integrantes do projeto se colocam no lugar do outro, podendo ser tanto destinatário como produto, com o intuito de se aproximar do problema.

A palavra chave dessa etapa é **pesquisa**. A pesquisa pode ser realizada por meio de entrevistas, conversas informais, buscas em livros e artigos, dentre outras fontes. Alguns autores gostam de dividir a etapa de pesquisa em dois momentos, a **pesquisa exploratória** e **pesquisa em profundidade**. A primeira, como o próprio nome explica, consiste em um momento em que o objetivo é compreender os aspectos básicos relacionados ao problema a ser enfrentado. A segunda, envolve um aprofundamento das informações identificadas na pesquisa exploratória, tentando identificar os principais aspectos e possíveis soluções, utilizando fontes como entrevistas, caderno de sensibilização, normalmente utilizado para obter informações de pessoas

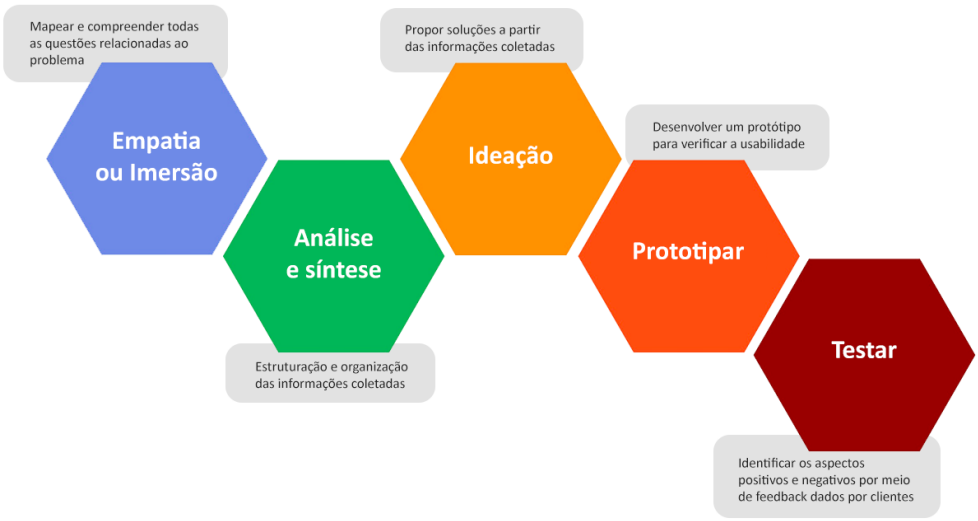

Figura 3.2: Imagem dos principais passos do *design thinking*

sem realizar um interferência em sua rotina, anotando aspectos práticos relevantes para a sua investigação, um dia na vida, esse utilizado pelo pesquisador para simular a rotina de determinado assunto ou pessoa estudada, dentre outras[6].

Aplicado ao processo de automação de documentos, trata-se da etapa em que o engenheiro ou arquiteto jurídico objetiva compreender a extensão do problema jurídico proposto por seu cliente. Nesse cenário, seria uma etapa para: (a) identificar os objetivos e propósitos que norteiam o negócio do cliente; (b) mapear todas as áreas de atuação do cliente, por exemplo, escritório *full service* que possui advogados atuantes na área tributária, trabalhista, cível, empresarial etc; (c) pesquisar no Poder Judiciário as principais demandas ajuizadas pelo escritório; (d) realizar entrevistas com advogados, sócios e gestores do escritório com o intuito de identificar os possíveis documentos jurídicos que poderiam beneficiar melhor o cliente com o processo de automação, dentre outros aspectos.

Análise e Síntese

A presente etapa possui como objetivo estruturar e organizar toda a informação coletada na etapa de imersão. Nesse cenário, é muito importante entender as informações encontradas, buscando sempre identificar padrões com o intuito de que as informações possam estar estruturadas de forma a auxiliar na resolução do problema.

Várias ferramentas e metodologias podem ser utilizadas para estruturar as principais informações encontradas a respeito do problema identificado, como: cartões de *insights*, que consistem em cartões com informações pontuais que auxiliam na anotação de pontos importantes identificados durante a pesquisa e permitem o rápido manuseio de tais informações; diagrama de afinidades, que basicamente envolve a organização de informações provenientes dos cartões de *insights* em forma de diagramas, indicando os principais temas correlacionados; mapa conceitual, é uma estratégia utilizada para que os dados coletados sejam visualmente organizados, auxiliando na compreensão de estruturação de dados complexos; critérios norteadores, consiste na identificação de critérios que irão conduzir a resolução do problema, dentre outras ferramentas e

[6]Para conhecer mais fontes acesse: https://www.youtube.com/watch?v=nKIu9yen5nc

metodologias [16].

Ideação

Realizada toda a pesquisa referente às informações necessárias para resolver o problemas, bem como, após estruturar tais informações de modo que façam sentido, passa-se à etapa de ideação. Essa etapa é voltada para propositura de soluções, em que todos envolvidos no projeto possuem a liberdade para compartilhar suas ideias, com o objetivo de chegar mais próximo de um produto ideal.

Aqui, a criatividade é a palavra chave. Este é o momento para pensar em ideias inovadoras para solucionar o problema proposto. Também podem ser utilizadas ferramentas e metodologias para estimular e estruturar tais ideias. Dentre as diversas metodologias a serem utilizadas podemos citar, a mais conhecida, o *brainstorming*. Trata-se de uma técnica voltada para estimular a produção do maior número de ideias, no menor tempo, sem o compromisso de delimitar uma solução específica. Em poucas palavras, o intuito é explorar diversas alternativas para solucionar o problema.

Abaixo, segue, um exemplo de ferramentas que podem auxiliar na elaboração do *brainstorming*.

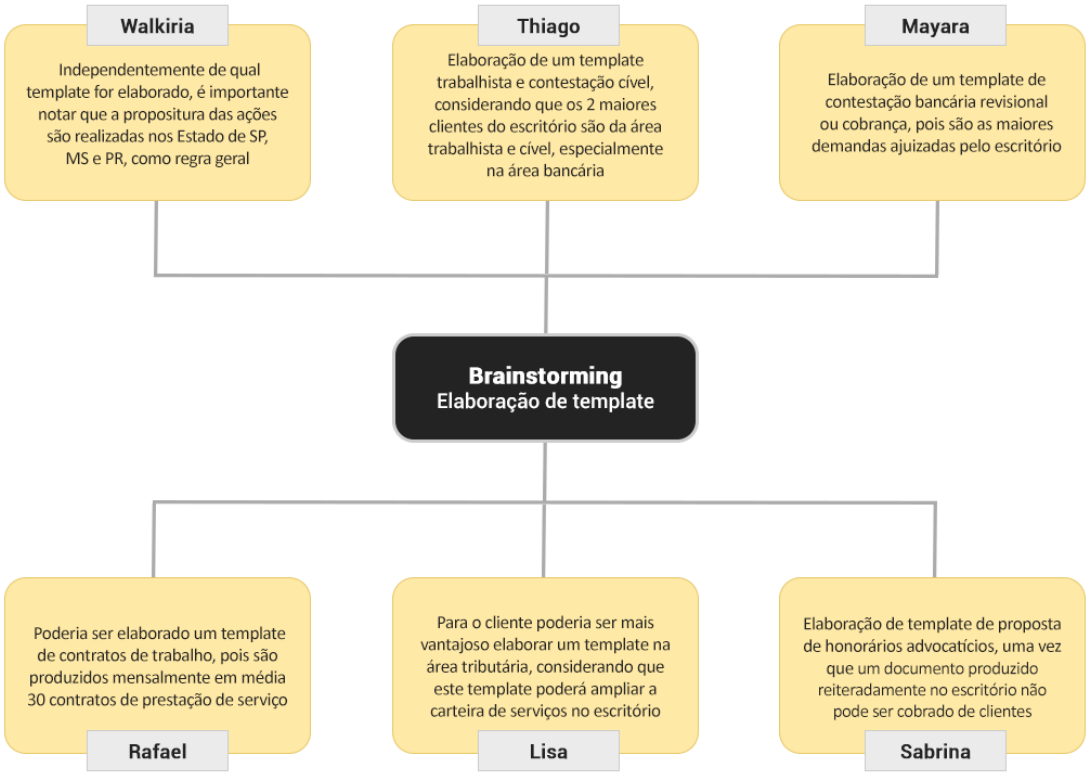

Figura 3.3: Mapa conceitual

A Figura 3.3 mostra o resultado de uma reunião de *brainstorming*, onde se discutiu tipos de documentos a serem produzidos para um documento específico, isto é, possibilidade de documentos a serem automatizados. Na Seção 3.4 abordaremos esse tema novamente com maior profundidade.

Prototipação

A prototipação é a etapa em que será validada a alternativa que foi escolhida entre as ideias sugeridas para a resolução do problema, é o momento "mão na massa". Com todas as informações organizadas, estruturadas e debatidas, objetiva-se criar um protótipo que possa interagir com o cliente agregando valor.

Muito embora esta etapa esteja alocada ao final da resolução do problema, pode ser desenvolvida concomitantemente durante outras etapas. Assim, ao longo da produção do documento as ideias podem ser testadas e inclusive implementadas em caso de validação positiva fornecida pelos clientes.

Ainda utilizando o exemplo dado anteriormente no processo de automação, com base nas informações coletadas e as ideias que surgiram durante o processo de *brainstorming*, optou-se por elaborar um protótipo de uma contestação cível, sendo considerada relevante ao cliente uma contestação revisional bancária. Ademais, é importante ressaltar que durante a pesquisa e entrevistas com os gestores, verificou-se que a atuação do cliente estava voltada para defesa de dois bancos, por isso, o *template* deveria considerar os temas que são abordados em 95% das ações ajuizadas pelos bancos, como exemplo: (a) exoneração de garantia, (b) ilegalidade de cláusulas e encargos, (c) inscrição indevida em cadastro de inadimplentes, (d) invalidade e/ou ilicitude do empréstimo consignado, (e) ocorrência de cobrança indevida, (f) venda casada de contrato bancário com cláusula de seguro etc. Além disso, as jurisprudências aplicáveis nos Estados de São Paulo, Mato Grosso do Sul e Paraná também deveriam ser abordadas.

Teste

Esta etapa final é voltada para a realização de testes com o intuito de ajustar pontos que não foram contemplados ou não são adequados ao produto desenvolvido. Esse é um momento muito importante, pois muitos ajustes são identificados apenas na efetiva utilização do protótipo, por isso, o teste e validação pelo cliente ou usuário final é fundamental.

No âmbito da automação de documentos jurídicos, ao estimular que os operadores do Direito usem o *template* produzido é que se descobre a ausência ou necessidade de inserir alguns pontos. No caso da contestação revisional bancária, apenas na realização de testes é que foi possível identificar, por exemplo, a ausência de alguns temas que normalmente são abordados na inicial, como pedido de danos morais ou mesmo suposta infração de princípios e regras jurídicas abstratos, temas que não foram mapeados anteriormente.

Assim como na etapa anterior, o teste pode ser aplicado em outros momentos da produção, sendo importante o *feedback* dos seus clientes e usuários sobre o produto criado, aproximando-se de um produto ideal.

3.2 Métodos Ágeis

Para os profissionais que atuam com o Direito "métodos ágeis" normalmente não fazem parte de nosso vocabulário. No entanto, com o desenvolvimento de novas competências e habilidades decorrentes da interação com a tecnologia e como os serviços jurídicos são prestados, tais preceitos e objetivos naturalmente passaram a fazer parte da nossa rotina, nos auxiliando no processo de produção e desenvolvimento de produtos jurídicos. Para tanto, é importante compreender o que são métodos ágeis e as principais metodologias utilizadas no mercado.

A concepção de métodos ágeis surgiu em 2001, por um grupo de 17 (dezessete) pessoas que se reuniu para pensar em formas de aprimorar o desenvolvimento de *software*, possuindo como premissa a construção de um modelo de negócios, permeados por confiança e respeitos pelos membros, baseados na colaboração de pessoas e a construção de comunidades em que tais membros tivessem interesse de trabalhar. Essa premissa foi o "pano de fundo" para a criação dos 12 (doze) princípios, conhecidos como o Manifesto Para Desenvolvimento Ágil em Softwares[7].

Os princípios do Manifesto Para Desenvolvimento Ágil em Softwares, trazem premissas importantes para o desenvolvimento de softwares, como:

(a) **1º Princípio**. Satisfação de clientes, por meio de entrega adiantada e contínua de *software* de valor;

(b) **2º Princípio**. Adaptação às mudanças de requisitos, com o propósito de que nossos clientes possam obter vantagens competitivas;

(c) **3º Princípio**. Promover um desenvolvimento contínuo de *software*;

(d) **4º Princípio**. Desenvolver o trabalho em conjunto entre pessoas relacionadas aos negócios e desenvolvedores;

(e) **5º Princípio**. Construção de projetos por indivíduos motivados;

(f) **6º Princípio**. Transmissão de informações para o time de desenvolvimento, por meio de uma conversa cara a cara;

(g) **7º Princípio**. Garantir que o *software* funcione é considerada uma medida primária de progresso;

(h) **8º Princípio**. Manter um ambiente sustentável, promovendo a passos constantes de patrocinadores, desenvolvedores e usuários;

(i) **9º Princípio**. Desenvolver a excelência técnica e bom *design*, aumentando a agilidade do projeto;

(j) **10º Princípio**. Pautar-se pelo princípio da simplicidade, em que a arte de maximizar a quantidade de trabalho que não precisou ser feito;

(k) **11º Princípio**. Estimular times auto-organizáveis, pois decorrem melhores arquiteturas, requisitos e *designs*; e

(l) **12º Princípio**. Promover, regularmente, uma reflexão sobre o como os times podem ser mais efetivos e realizarem ajustes para incidir em tal prática.

A partir desses princípios é possível identificar que os métodos ágeis visam estimular o desenvolvimento de *software* com foco na comunicação entre as partes envolvidas no processo em um ambiente colaborativo, bem como a flexibilidade para fácil adaptação em caso de modificações necessárias durante o processo de desenvolvimento. Além disso, estimulam a agilidade no desenvolvimento sempre realizando entregas, mesmo que pequenas, e também foco na entrega de produtos com qualidade e excelência técnica. Estimula, ainda, o aprimoramento do produto durante e após o seu desenvolvimento.

É importante notar que esses métodos surgiram em resposta aos gargalos enfrentados no desenvolvimento de produtos tecnológicos, em especial a instabilidade relacionada à definição e escopo do produto a ser

[7]Para mais informações sobre o Manifesto Para Desenvolvimento Ágil acesse: http://agilemanifesto.org/

desenvolvido, pois muitas vezes os clientes não conseguem definir os requisitos mínimos a serem considerados no projeto.

Para nós, engenheiros e arquitetos jurídicos, funcionam como "boas práticas" a serem observadas durante o processo de elaboração de um *template* jurídico. E abaixo se encontram alguns dos métodos ágeis mais utilizados no desenvolvimento de produtos.

Scrum

O Scrum é uma metodologia usada para gerenciar e desenvolver produtos complexos. Conforme definido por seus criadores, trata-se de:

> Um *framework* dentro do qual pessoas podem tratar e resolver problemas complexos e adaptativos, enquanto produtiva e criativamente entregam produtos com o maior valor possível.

O *framework* Scrum não envolve apenas o processo de desenvolvimento do produto, mas sim, uma forma de organização e estruturação de trabalho (talvez, esse é o motivo pelo qual se denomina *framework*) que contempla times e os papeis de seus integrantes, regras durante o processo de desenvolvimento, eventos formais para inspeção e adaptação do produto e a efetiva produção de artefatos.

Com relação aos times e seus colaboradores, a ideia é que se tenha um pequeno time de pessoas. De acordo com o *framework* Scrum, grupos menores são propensos à uma melhor flexibilidade e a capacidade de se adaptarem mais rapidamente às novas demandas, estimulando uma alta produtividade no ciclo de desenvolvimento de produtos.

Além disso, podemos dividir as responsabilidades dos integrantes do grupo em *product owner*, time de desenvolvimento e *scrum master*. O *product owner* ou como a própria tradução sugere "dono do produto" é o ente responsável pelo gerenciamento da especificação do produto, organizando os itens de *Backlog* do produto, que em português pode ser entendido como "itens a serem realizados" no processo de desenvolvimento do produto. Nesse sentido, possui a função de (a) organizar as tarefas que deverão ser cumpridas pelo time de desenvolvimento com o propósito de alcançar mais agilidade e eficiência e (b) garantir que os integrantes do time de desenvolvimento compreendam de forma clara as tarefas a serem cumpridas.

O time de desenvolvimento é composto por integrantes que são propriamente os executores e responsáveis diretos pelo desenvolvimento do produto. Normalmente o time de desenvolvimento é estruturado para gerenciar o seu próprio trabalho, sendo que cada integrante possui habilidades diferentes que contribuem à equipe e o processo de produção. É importante mencionar que não existe uma divisão interna, restando todos aptos a atuarem na realização de testes, arquitetura e estruturação.

Por sua vez, o Scrum Master é o responsável por acompanhar e implementar o *framework* Scrum, auxiliando aos demais na compreensão teórica e aplicação prática do Scrum. Nesse sentido, possui um papel de "facilitador", promovendo diálogo próximo com o *product owner* e os integrantes do time de desenvolvimento.

Os eventos formais no *framework* Scrum são destinados à criar momentos específicos para projetar e organizar o desenvolvimento do projeto, podendo estabelecer metas, objetivos de trabalho semanal ou proporcionar a oportunidade de ajustar algum ponto durante o desenvolvimento. É uma ferramenta eficaz para minimizar

reuniões prolongadas e com pouca eficiência. Assim, os eventos formais podem utilizar-se da Sprint e reuniões diárias para a equipe.

A *sprint* consiste no tempo determinado para cumprir metas específicas, quebrando em etapas a construção e o desenvolvimento do trabalho. Normalmente são realizadas reuniões de *sprint* para que sejam definidos os objetivos e propósitos a serem cumpridos naquele espaço de tempo, podendo ser realizados semanalmente, quinzenalmente ou no máximo, mensalmente. O objetivo é garantir a execução de metas e, caso aplicável, a realização de ajustes durante o processo de desenvolvimento.

Por sua vez, a reunião diária é voltada para os integrantes do time de desenvolvimento para que todos possam informar a evolução de suas tarefas previstas na *sprint*, bem como identificar como podem colaborar com os demais colegas para cumprir os objetivos propostos na Sprint. Em linhas gerais, trata-se de um alinhamento de expectativas e planejamento para o próximo dia das atividades a serem executadas. Nessas reuniões a regra é produtividade, por isso, não podem ultrapassar o período de 15 (quinze) minutos. Essas reuniões diárias são instrumentos importantes para a facilitar o canal de comunicação entre os integrantes da equipe.

Por fim, a produção de artefatos, que podem ser considerados como "pedaços" do produto, são projetados para alinhar e promover a transparência da construção do produto final, podendo ser objeto de ajustes e readequação em seu desenvolvimento.

Em suma, esses são os principais pontos abordados em um *framework* Scrum. Para mais detalhes e informações específicas, recomenda-se a leitura do Guia Scrum escrito por Schwaber e Sutherland[8].

Kanban

O Kanban é uma metodologia focada para organizar e gerenciar o fluxo de trabalho. Nas palavras de Janice Linden-Reed presidente da organização Lean Kanban, "é um método que demonstra como é o nosso fluxo de trabalho". Uma metodologia que se utiliza da transparência para demonstrar a realidade, possibilitando a identificação de pontos que podem ser melhorados e aprimorados ao longo do fluxo de trabalho.

A metodologia foi desenvolvida por David J. Anderson e denominada como método em 2007, período em que Anderson começou a ministrar apresentações sobre os métodos que estava utilizando na Microsoft. Em 2010, conjuntamente outros entusiastas sobre a metodologia é lançado o livro "Kanban - Successful Evolutionary Change for your Technology Business" se tornando a referência de métodos e ferramentas no âmbito da metodologia Kanban.

O Kanban parte de algumas premissas, a seguir retratadas:

(a) **Metodologia especializada**. Cada instituição, organização, empresa ou escritório possui suas peculiaridades, por isso, não seria adequado aplicar uma solução única para todos esses entes. Nesse sentido, estimula-se o auto conhecimento de cada instituição, em especial no seu processo de desenvolvimento;

(b) **Transparência no processo de desenvolvimento**. A transparência fornece conhecimento acerca da performance no desenvolvimento do trabalho, com foco em mensuração e validação do processo. Assim, fornece à instituição um conhecimento prático sobre o processo de trabalho, estimulando a previsibilidade em sua atuação;

[8] O Guia do Scrum. Um gia definitivos para o Scrum: As regras do Jogo acesse: www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-Portuguese-Brazilian.pdf

(c) **Promoção de evoluções no método de trabalho**. A ideia é promover a reflexão acerca do processo de trabalho de determinada instituição e, assim, identificar os pontos que precisam ser ajustados. A finalidade é realizar pequenos ajustes e modificações, justamente para que as mudanças sejam estimuladas e aceitas, sem provocar rejeições. Aqui a ideia é estimular um evolução no processo e não uma revolução;

(d) **Comprometimento com agilidade**. Assim como outros métodos ágeis, o Kanban propõe práticas que busquem agilizar o fluxo de trabalho;

(e) **Método em construção**. Trata-se de um método em constante transformação. A todo momento são desenvolvidas pela comunidade da metodologia Kanban novas técnicas que aprimoram e aperfeiçoam essa metodologia. É um projeto em construção!;

(f) **Método de gerenciamento de risco**. Também é utilizada como uma ferramenta de gerenciamento de riscos, considerando que sua atenção é voltada para a realização de *feedbacks* e mensuração e validação do processo, sendo um mecanismo preciso na identificação de áreas de risco;

(g) **Equilíbrio entre demanda e capacidade**. O Kanban utiliza ferramentas que possuem a finalidade de aprimorar como as requisições de trabalho são tratadas, maximizando a produtividade e eficiência;

(h) **Aplicabilidade em outras áreas**. Muito embora essa metodologia possa se enquadrar e auxiliar na produção e desenvolvimento de produtos voltados à tecnologia, ela pode ser utilizada em outras áreas como educação, Direito, produção de filmes etc;

O trabalho em equipe na metodologia Kanban se utiliza de um quadro para proporcionar uma melhor visualização do fluxo de trabalho da equipe. Esse quadro pode ser físico ou virtual[9] para o gerenciamento das atividades. Abaixo está um exemplo simples de organização de fluxo de trabalho:

Essas são as ideias gerais que fundamentam a utilização da metodologia Kanban. Para um aprofundamento recomenda-se a leitura do livro Essential Kanban Condensed escrito por David J. Anderson e Andy Charmichael[10].

3.3 Programação Extrema (XP)

É uma metodologia de desenvolvimento de *software* produzida para melhorar a qualidade do *software* e se destaca pela habilidade em se ajustar rapidamente às demandas solicitadas pelos clientes. Essa metodologia foi desenvolvida em meados dos anos 90 por Ken Beck enquanto trabalhava na Chrysler e desenvolvia um sistema de gerenciamento de pagamento da companhia. Em 1999, Beck publicou o livro Extreme Programming Explained, detalhando sua metodologia para o público em geral, repercutindo em grande sucesso.

A metodologia de programação extrema possui cinco valores que norteiam o desenvolvimento de *software*, que são: simplicidade, comunicação, *feedback*, respeito e coragem. **Simplicidade** na condução e desenvolvimento do projeto, executando apenas o que for solicitado. A **comunicação** é um fator importante, considerando que todos fazem parte da equipe e deverão trabalhar conjuntamente para a execução das tarefas. O **feedback**

[9]Existem várias ferramentas de gestão e trabalho em equipe virtuais que podem auxiliar nesse processo como (a) Trello, acesse em: - https://trello.com e (b) Jira, acesse em: https://br.atlassian.com/software/jira, dentre outros.

[10]O livro Essential Kanban Condensed está disponível gratuitamente em: http://leankanban.com/wp-content/uploads/2016/10/Essential-Kanban-Condensed-7-28-2016.pdf.

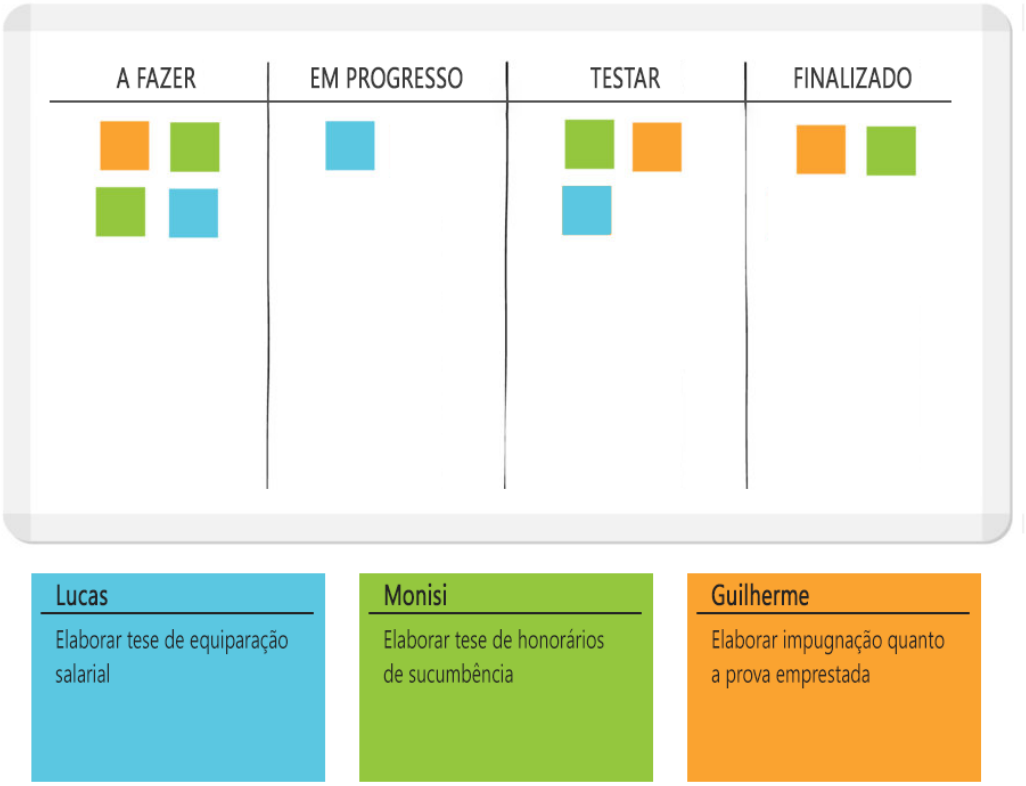

Figura 3.4: Exemplo de quadro Kanban

é muito importante para o desenvolvimento do produto, sendo passível de quaisquer ajustes solicitados. O processo de desenvolvimento do produto pode ser ajustado e não o contrário! O **respeito** é fundamental, a ideia é que todos os integrantes da equipe merecem respeito e devem ser valorizados, possuindo a liberdade de contribuir com o projeto. Por fim, **coragem** para dizer a verdade sobre o progresso e estimativas. Não há espaço para medo, pois ninguém trabalha sozinho!

A programação extrema envolve a utilização de algumas regras para nortear o processo de desenvolvimento do produto, passando por cinco etapas, sendo planejamento, gerenciamento, *design*, codificação e teste.

Na etapa de planejamento, como se auto explica, o objetivo é planejar cada etapa de desenvolvimento. Assim, estimula-se a anotação de requerimentos solicitados pelo cliente, técnica denominada *user stories*. Após, é o momento de elaborar um plano em que serão determinados quais requerimentos serão implementados para cada sistema produzido e as respectivas datas. Essa técnica utilizada é denominada de *release plan*, que traduzindo para o português significa "plano de entrega". É nessa etapa que se realiza toda a estratégia de desenvolvimento do produto, sendo que a equipe de desenvolvedores, gerentes do projeto e os clientes podem decidir conjuntamente para estabelecer a primeira entrega e as possíveis estimativas de prazo de produção para entregas futuras. É muito importante que as entregas dos produtos sejam realizadas constantemente, sendo o projeto dividido em iterações com o intuito de facilitar o desenvolvimento do produto. A cada iteração, a ideia é que o produto já tenha sido testado e tenha condições de ser demonstrado aos clientes, sendo passível de ajustes.

Na etapa de gerenciamento a ideia é organizar um ambiente propício para que a equipe de desenvolvedores possa trabalhar de forma produtiva e eficiente, realizando ajustes quando necessário. A comunicação é muito importante para estimular o trabalho em equipe, por isso, pequenos ajustes no ambiente como criação de

espaços abertos que demonstra a igualdade entre os membros da equipe se mostra relevante, bem como utilizar quadros que indiquem a evolução do projeto, de forma visual, também se mostra relevante.

Além disso, estimular reuniões de *stand up*, que consistem em reuniões diárias realizadas em pé, justamente para que seja rápida e produtiva, para falar sobre três pontos: (i) o trabalho do dia anterior, (ii) quais são os objetivos de trabalho do dia presente; e (iii) falar sobre eventuais problemas que estão atrapalhando a execução de sua atividade. A proposta do *stand up* é evitar o desperdício de tempo e produtividade em reuniões. Essa é uma metodologia utilizada muito pela Engenharia e Arquitetura Jurídica, vez que aproxima a equipe para dialogar, estimulando que esses atores produzam durante o dia e possam compartilhar sobre a sua atividade, bem como é o momento para verificar se alguém da equipe está passando por alguma dificuldade e precisa de ajuda.

Para que a equipe de desenvolvedores seja produtiva é muito importante que tenha uma rotatividade de tarefas. A rotatividade traz uma flexibilidade para os integrantes e, consequentemente, a equipe, que pode executar quaisquer tarefas. Ao invés de acumular atividades em alguns membros de equipe todos terão condições de contribuir igualmente, tornando a equipe mais produtiva.

Na etapa de *design* a ideia central é **simplicidade**. Aqui a finalidade é utilizar a simplicidade nas diversas etapas do projeto. Especialmente no que diz respeito à codificação, recomenda-se utilizar o **Testable, Understandable, Browsable e Explainable - TUBE**, que traduzido para o português significa um código passível de **teste**, **compreensão**, **localização** e **explicação**. O teste consiste na habilidade de desenvolver mecanismos de testes de unidade e aceitação que possam rapidamente verificar problemas, ou seja, deixar a codificação estruturada de forma que pequenos testes podem ser realizados ao longo da produção. A compreensão consiste no ato de se desenvolver um código que outras pessoas que não estão ligadas ao projeto possam compreendê-lo. Já a localização, consiste na estruturação de um código de qualidade, fazendo com que qualquer pessoa consiga achar o que precisa a qualquer momento. Por fim, desenvolver um código explicativo é aquele que possa ser apresentado a qualquer pessoa.

Já na etapa de codificação há um esforço coletivo para concretizar o projeto. Ter o cliente próximo e sempre disponível durante o desenvolvimento do produto é fundamental, pois diminui o risco de entregar produtos que não satisfaçam os anseios do cliente, fazendo perder tempo para modificar e refatorar o código. Essa é uma dica de ouro! O quanto antes o cliente for incluído no processo de estruturação e desenvolvimento do produto, maior será a probabilidade do cliente ter aderência ao produto (ele se sente parte importante no processo de construção do produto), "cliente presente é sinônimo de cliente contente".

Além disso, considerando que muitos integrantes vão participar da codificação do produto é importante estabelecer critérios e boas práticas de codificação para que todos envolvidos no projeto possam elaborar um código consistente e fácil para trabalhar e, caso necessário, refatorar.

A etapa de testes, é auto explicativa, pois possui o propósito de fomentar testes ao longo do processo de desenvolvimento do produto. Testar é uma etapa importante na rotina do engenheiro jurídico, nenhum código pode ser entregue ao cliente sem que seja testado. A cada tarefa produzida é recomendável a realização de testes. Quanto antes forem identificados erros, que no âmbito da programação são denominados como "bugs", mais fácil será a realização de modificações. Esse é um hábito fundamental que deve ser cultivado pelos desenvolvedores.

A metodologia de programação extrema possui uma preocupação muito grande no quesito testes. Nesse sentido, estimula a realização de testes concomitantemente com o desenvolvimento do produto, ou seja,

a realização de testes é tão importante quanto o desenvolvimento do próprio produto. Assim propõe a realização de testes de unidade, integração e sistema. Os testes de unidade permitem a identificação de *bugs* em uma unidade isolada no código, auxiliando na verificação da usabilidade de determinado ponto específico. Esse tipo de teste é utilizado em todas as etapas de desenvolvimento do produto. Já o teste de integração é realizado em que módulos ou agrupamento de unidades que são combinados e testados em conjunto, também podendo ser realizado durante todas as etapas de desenvolvimento do produto. O teste de sistema é voltado para verificar a usabilidade do sistema como um todo.

Quanto às práticas utilizadas na metodologia de programação extrema, destacam-se:

1. **Versões Pequenas**: *Release* é uma parte do produto que deverá ser implantado no cliente, pronto para ser utilizado. As *Releases* são entregues em pequenos pedaços e com frequência. A ideia é realizar a entrega constante ao cliente, recebendo *feedback* imediato sobre o produto e realizar ajustes rápidos, caso necessário.

2. **Jogo do Planejamento**: Seu objetivo é delimitar o escopo da próxima *Release*, estipulando ordens e prioridades das sub-tarefas segundo demandas do negócio e estimativas técnicas. Outro fator interessante a se considerar é a participação do cliente nesta etapa do projeto. Os desenvolvedores definem estimativas, técnicas adotadas, e por fim os desenvolvedores ajudam no detalhamento de prazos para as atividades.

3. **Programação em pares**: Duas pessoas trabalhando em uma mesma máquina é um conceito estranho a muitos. No entanto, o processo de nivelamento de conhecimento é atestado. Normalmente, um critica ou dá sugestões, enquanto o outro codifica propriamente. Os pares trocam de lugar periodicamente. Essa prática é excelente e favorece comunicação e aprendizado.

4. **Testes automatizados** são práticas que garantem a robustez do sistema diante das mudanças que ocorrem nas especificação do projeto. Os testes de unidade, por exemplo, encapsulam uma parte do código e o testam individualmente, sem qualquer interferência de outra parte. Assim, dada qualquer mudança do código de outras partes do projeto, pode-se testar em poucos segundos e averiguar a corretude do trecho encapsulado.

5. **Refatoração**: A refatoração significa melhorar o código sem alterar qualquer funcionalidade sua. Para facilitar a realização de mudanças é sempre recomendável a refatoração do código. A refatoração contínua possibilita manter um bom projeto, apesar das mudanças frequentes.

6. **Propriedade Coletiva**: Códigos não pertencem somente a uma pessoa. A propriedade coletiva é a melhor forma de evitarmos problemas como trocas de membros da equipe.

7. **Padrões e boas práticas de Codificação**: Conforme dito, o código é propriedade coletiva. Assim é mister que mantenhamos um padrão de qualidade elevado para obter resultados satisfatórios. Assim, todos da equipe conseguem interagir com o código, entender minúcias técnicas e sentirem-se capazes de modificar, adicionar e refatorar códigos escritos por outras pessoas.

8. **Integração Contínua**: O princípio por trás de integração contínua é que todo código deva ser integrado diariamente e todos testes automatizados devam responder afirmativamente antes e depois da integração com as demais partes do código. Se algum problema é encontrado ele deve ser corrigido antes da integração.

9. **Metáfora**: é um bom recurso comunicativo que rompe barreiras de compreensão de uma ideia abstrata. O uso de metáforas é considerada fundamental para transformar abstrações em aplicações práticas.

Na prática do processo de transformação digital de conteúdo jurídico não usamos apenas uma metodologia, escolhemos as principais ferramentas e métodos utilizados em cada metodologia, adequando sempre ao trabalho que será prestado, bem como aos valores dos integrantes de cada equipe para estimular o máximo de produtividade e eficiência.

3.4 Ferramentas de análise

Seguem algumas ferramentas para especificação de projetos de sistemas jurídicos, em especial, para automação de documentos.

Brainstorming

O conceito de *brainstorming* - introduzido pelo publicitário Alex Osborn em 1948 - é um processo criativo no qual um grupo de pessoas se reúne para apresentar soluções, ideias e reflexões a respeito de um problema proposto. Ainda que as ideias possam parecer absurdas ou muito simples, todas as ideias são colocadas a mesa. É necessário respeitar o "momento criativo" e deixar fluir qualquer tipo de pensamento, por mais absurdo que possa parecer. O objetivo do método é que ao final, após a "tempestade de ideias", sejam selecionadas as melhores estratégias para compor tanto o planejamento, quanto a execução do projeto.

Muito embora a técnica seja muito bem estabelecida no mercado publicitário, o *brainstorming* se apresenta como uma ferramenta muito útil em outros campos, em particular, na esquematização de projetos de automação. Por exemplo, o *brainstorming* pode ser utilizado para detectar quais entidades e objetos estão presentes em determinado modelo conceitual ou peça jurídica, para identificação de escopo de um projeto de automação, para melhoria de um procedimento de captação de requisitos de projeto, entre outras aplicações.

Em linhas gerais, a efetividade do *brainstorming* é sustentada pela observação de pontos essenciais [17, 18], os quais destacam-se:

- **Defina o problema:** Embora pareça óbvio esse item, na pratica não é! O objetivo principal do *brainstorming* é resolver problemas. O único pré-requisito suficiente para se adotar essa técnica é que exista um problema a ser resolvido. Não perca esse objetivo! Caso contrário, o *brainstorming* poderá se tornar uma palestra desinteressante ou simplesmente um jogo de ideias sem objetivos concretos. Apresente o problema de forma clara e precisa.

- **Prepare previamente o encontro:** Um dos pontos fortes do *brainstorming* é a espontaneidade que as ideias surgem. Entretanto, tome cuidado! Não confunda espontaneidade com improviso. O *brainstorming* é uma reunião como qualquer outra que precisa ser planejada, guiada e pensada! Existem muitas variantes de *brainstorming*, como por exemplo, *brainwriting* [19], onde os participantes escrevem e leem em vez de falar e ouvir. Obviamente, precisa-se planejar o encontro, criar um ambiente agradável, verificar se existe uma variante mais adequada ao grupo (caso existam membros amantes da dialética).

- **Use materiais de apoio:** A procura de uma solução em um ambiente de muitas ideias pode ser um problema e tanto. Sendo assim, tome nota de tudo, use material de suporte: cartões, *post-its*, lousa, *softwares*. Não se esqueça de documentar tudo! Mesmo as ideias mais triviais podem dar à luz uma ideia genial.

- **Modere os conflitos:** A essência do *brainstorming* é a pluralidade de ideias. E como se trata de seres humanos, os conflitos são eminentes. Saber administrá-los é uma tarefa essencial para garantir

a fluidez da reunião. Não tente dirimir os conflitos. Os conflitos são necessários. Apenas modere a discussão com equilíbrio e respeito. Sugira a construção de novos cenários a fim de explorar melhor o potencial criativo dos participantes.

- **Tente gerar soluções individuais:** Em seu artigo, Keeney [17] adverte para o efeito âncora, em que um participante pega carona na solução de um outro companheiro, convergindo as ideias para um universo de soluções óbvias e convencionais. Todos nós, seres humanos, tendemos a ouvir soluções de outros e estender as soluções apresentadas até um ponto que estagnamos. Isso ocorre na verdade, porque somente uma única ideia está sendo considerada. Suas variantes muitas vezes não alcançam envergadura suficiente pra resolver o problema de fato. Por isso, as ideias tidas inicialmente como muito simples podem calçar outras ideias complexas. Sendo assim, não interrompa o fluxo criativo de ninguém. É melhor jogar a ideia proposta fora depois, do que descartá-la de cara. Nunca exclua possibilidades! Não busque somente o convencional.
- **Concretude na solução:** Após se mostrar o esgotamento de ideias, organize-as e selecione aquelas que se mostraram mais efetivas. Formalize a resolução do problema. Trace o *road map* dos próximos passos. Delegue responsabilidades e, se possível, saia com esboços, diagramas e anotações concretas. A elaboração do mapa mental (*mind mapping*) é um exemplo de algo concreto sobre o qual o arquiteto pode estruturar seu raciocínio.

Brainstorming é um tema amplamente difundido. Há quem defenda. Há quem ataque. O fato é se mostra uma boa ferramenta para compor a caixa de ferramentas do arquiteto jurídico.

Mind Mapping

O *mind mapping* é uma técnica eficiente para transmitir e capturar informações de nossas mentes. É um mecanismo lógico-criativo baseado na representação do conhecimento através de um diagrama conhecido como mapa mental, ou *mind map*.

O *mind map* é um diagrama usado para visualizar informações organizadas por relações hierárquicas. Essas hierarquias exibem relacionamentos entre ideias associadas, as quais podem ser representadas por imagens, palavras, fragmentos de ideias etc.

O modelo tradicional de *mind map* exibe **conceitos** descritos dentro de círculos (sejam palavras ou imagens). O conceito central da ideia é posicionada ao centro do diagrama. Ideias fundamentais são conectadas diretamente a ideia central, assim como ideias secundárias são conectadas às fundamentais, e assim, sucessivamente, até compor todo o mapa. Quanto mais central a ideia, maior o destaque do círculo que a representa. As conexões entre os círculos, normalmente são feitas com arestas tão grossas quanto forem as conexões entre os conceitos ali representados.

O aspecto de um *mind map* se assemelha a raízes de uma árvore frondosa (ver Figura 3.5). É comum usar imagens ou formas pra destacar pontos de interesses ou outras particularidades relevantes à ideia central. A capacidade de representar relatórios informacionais (muitas vezes incompreensíveis) em um diagrama colorido, organizado e facilmente memorizável é uma das inúmeras vantagens do mapa mental.

Acredita-se que o conceito surgiu em meados do século III, com esboços do filósofo Porfírio de Tiro. Entretanto, credita-se a criação do *mind map* como o vemos hoje ao escritor Tony Buzan [20]. Buzan propagou seus conceitos de árvores radiais nas quais se destacavam palavras-chaves que conectavam entre si com um sistema de cores, de modo a compor um diagrama no formato de árvore. Um *mind map* se propõe a ser

muito mais que uma diagramação técnica: é uma ferramenta visual destacada para representar os associações lógicas entre conceitos. Observe um exemplo de *mind map*:

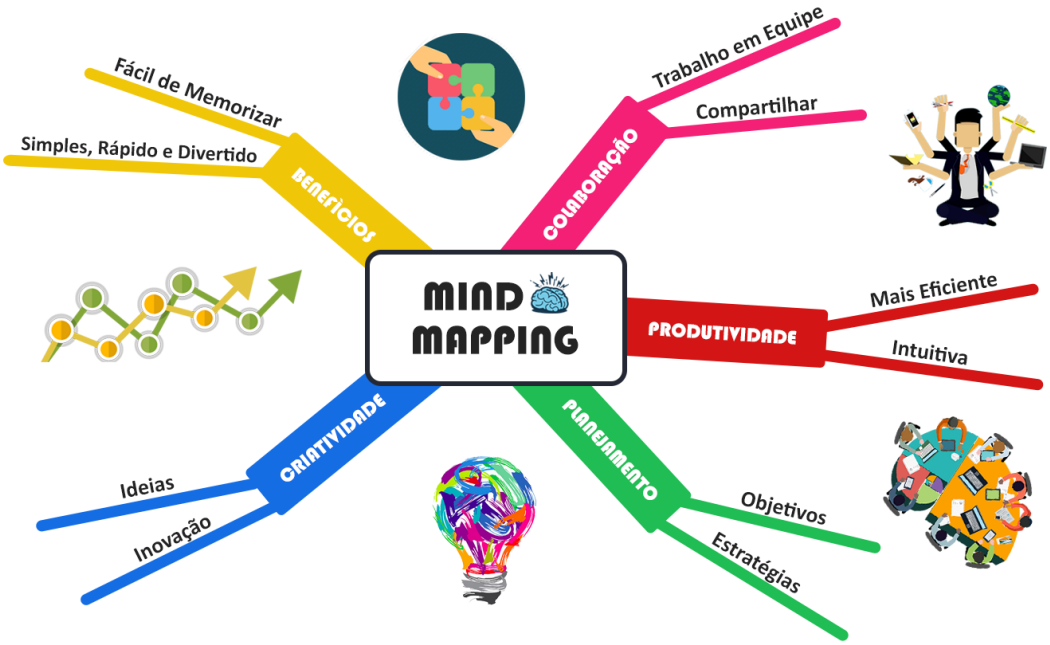

Figura 3.5: Exemplo de um *mind map* que representa os benefícios da prática do mesmo.

A Figura 3.5 é um *mind map* que representa os benefícios que traz o bom uso da ferramenta. Observe que o *mind map* possui a propriedade de ser auto explicativo. É quase um insulto à inteligência alheia explicar que o *mind mapping* apresenta benefícios, aguça a criatividade, desenvolve o espírito colaborativo da equipe, impulsiona a produtividade e favorece o planejamento estratégico de uma corporação.

Se pararmos para pensar, um *mind map* pode ser comparado ao sistema circulatório do corpo humano. O coração é o centro do mapa, que bombeia o sangue (informações) pelas artérias principais até os grandes órgãos do corpo (cérebro, fígado, pulmão etc). Os órgãos por sua vez, possuem veias que funcionam como propagadoras secundárias, as quais levam o sangue às partes mais internas do órgão. Os vasos sanguíneos por sua vez, levam menos sangue aos tecidos mais internos e assim por diante.

O *mind map* funciona como um espelho de nosso pensamento natural, representado por uma linguagem gráfica poderosa, que provê a chave universal para desbloquear a dinâmica potencial do cérebro. Segundo Buzan [20], existem alguns procedimentos básicos que podem guiar a criação de um *mind map*:

- Comece posicionando uma imagem (ou palavras) com bastante cor ao centro do mapa. Tal imagem corresponde ao tópico central do *mind map*.
- Use ao redor da imagem central, imagens, símbolos, códigos e crie várias dimensões no mapa. Conceitos contíguos também devem ficar próximos no mapa.
- Escreva algumas palavras-chaves e pequenos comentários, destacando-os com letras maiúsculas, minúsculas, negritos etc.
- Cada palavra ou imagem deverá estar conectada com outras, todavia, conservando suas próprias ramificações específicas.

- As linhas devem ser conectadas a partir da imagem central. Em geral, as linhas começam bem grossas e vão afinando na medida em que se distancia radialmente do centro.
- Pondere as linhas de acordo com a força do relacionamento. Ou seja, relacionamentos fracos (ou fortes) entre conceitos, use linhas finas (ou grossas).
- Use múltiplas cores para aguçar o estímulo visual e representar grupos ou classes dentro do *mind map*.
- Desenvolva seu próprio estilo de *mind map*.
- Enfatize associações e conceitos dentro do *mind map*.
- Procure manter seu *mind map* claro! A visualização deve ser limpa e clara. Costuma-se usar hierarquias radiais e ramificações de hierarquias externas para caracterizar os conceitos.

Lembre-se, a ideia, o sujeito ou o foco principal devem estar bem claros e caracterizados no centro do mapa, normalmente com uma imagem. Os temas principais irradiam (como feixes solares) desde a imagem central até as extremidades da ramificação. Cada tema (e subtema) é comumente representado por um ramo, mas por conta de *softwares* de diagramação, é conveniente representá-los por nós dentro da árvore. Tais nós aparecem na forma de imagens, palavras-chaves ou símbolos com suas respectivas linhas de associação. Tópicos de menor importância são representados por pequenas ramificações nas extremidades dos ramos maiores. Use cores para classificar elementos e ideias comuns.

Embora existam ferramentas bem parecidas com *mind map*, conceitualmente se diferem bastante em notação e propósito. Por exemplo, os mapas conceituais são uma versão bem simplificada de *mind map*, onde o foco está apenas sobre as palavras ou ideias. Em mapas conceituais não existem a figura central de uma entidade principal que governa todas as outras. O mapa conceitual é útil para detectar as entidades básicas de um documento jurídico, por exemplo.

Outros modelos baseados em grafos, tais como árvores de decisão, diagramas de entidade relacionamento, também são apropriados para lidar com uma modelagem jurídica, os quais veremos mais adiante. No entanto, os grafos são mais gerais, formais e usam menos recursos mnemônicos[11] que os *mind maps*.

Vamos a um exemplo de aplicação no direito: Considere que queremos montar o *mind map* de uma contestação trabalhista padrão. Segue a receita de bolo para montar o seu:

1. Defina qual o tema principal de seu *mind map* e escreva-o no centro da página. Embora pareça óbvia,

Figura 3.6: Etapa 1. Escolhendo o tema central.

essa tarefa precisa ser cuidadosamente preparada. Muitas vezes o problema a ser questionado é apenas um efeito do real problema a ser atacado. No nosso exemplo, ilustraremos um *mind map* bem básico cujo tópico central é a própria contestação em si mesma.

[11] Para fins deste trabalho, recursos mnemônicos consiste na utilização de imagens e elementos que auxiliam associações rápidas e memorização.

2. Descubra subtemas relacionados ao conceito central e os desenhe próximos a ele, dispostos de forma radial (preferencialmente). Grandes ramos (linhas) devem ligar tais subtemas ao conceito central. Use frases curtas, no máximo três palavras para descrever os conceitos e os subtemas. No nosso exemplo,

Figura 3.7: Etapa 2. Definindo os subtemas relacionados.

existe o enfoque em quatro temas principais: as partes, os advogados, o endereçamento da contestação e os temas contestados.

3. Tente pensar em pelo menos dois ou três pontos principais para cada subtema criado e expanda ramificações a partir deles. Usar cores variadas para segmentar conceitos é também uma boa prática. Observe como se desdobraram os tópicos principais em subtemas no exemplo da contestação. É claro que o exercício de desdobramento pode se estender muito mais e compor um cenário bem mais completo da contestação. Fica a critério do leitor continuar esse desdobramento.

4. Adicione imagens, símbolos ou outros elementos que invocam pensamentos ou mensagens através deles.

Um dos grandes problemas da composição de um mapa mental é a definição do escopo do trabalho. Manter o escopo amplo e detalhado o suficiente não é algo trivial de se fazer, uma vez que qualquer tema possui profundidade e abrangência tão grande quanto se queira. A tarefa do projetista nesse caso é manter os pés no chão e apresentar uma proposta de trabalho realista e ao mesmo tempo substancial.

Existem muitos *softwares* dedicados à produção de *mind maps*, dentre os quais destacamos o Lucid Chart (`www.lucidchart.com`), Bubbl (`https://bubbl.us`), MindMup (`www.mindmup.com`), entre outros. Existem também *softwares* especializados em realizar reuniões de *mind mapping online*. Vale a pena conferir. ;-)

Exercício 1. Agora que entendemos o que *brainstorming* e *mind mapping* são, vamos exercitar esses conceitos no mundo jurídico! Desenhe um mapa mental do processo de elaboração de uma peça contratual de seu interesse. Abuse de sua criatividade, use muitas anotações, extraia o máximo de conceitos do problema que você puder. Mas atenção: mantenha seu *mind map* limpo!

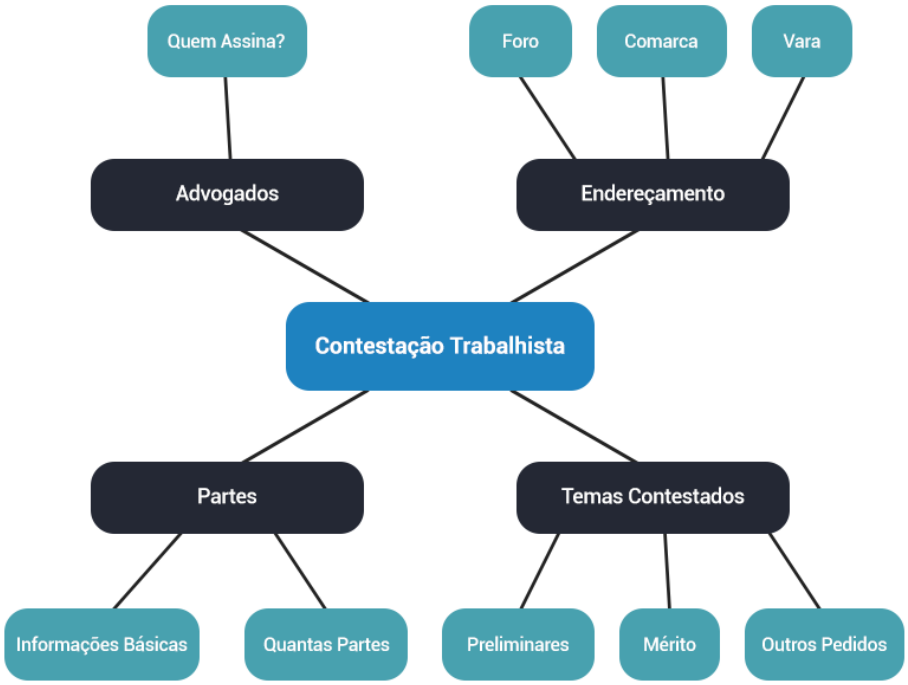

Figura 3.8: Etapa 3. Refinando tópicos de subtemas.

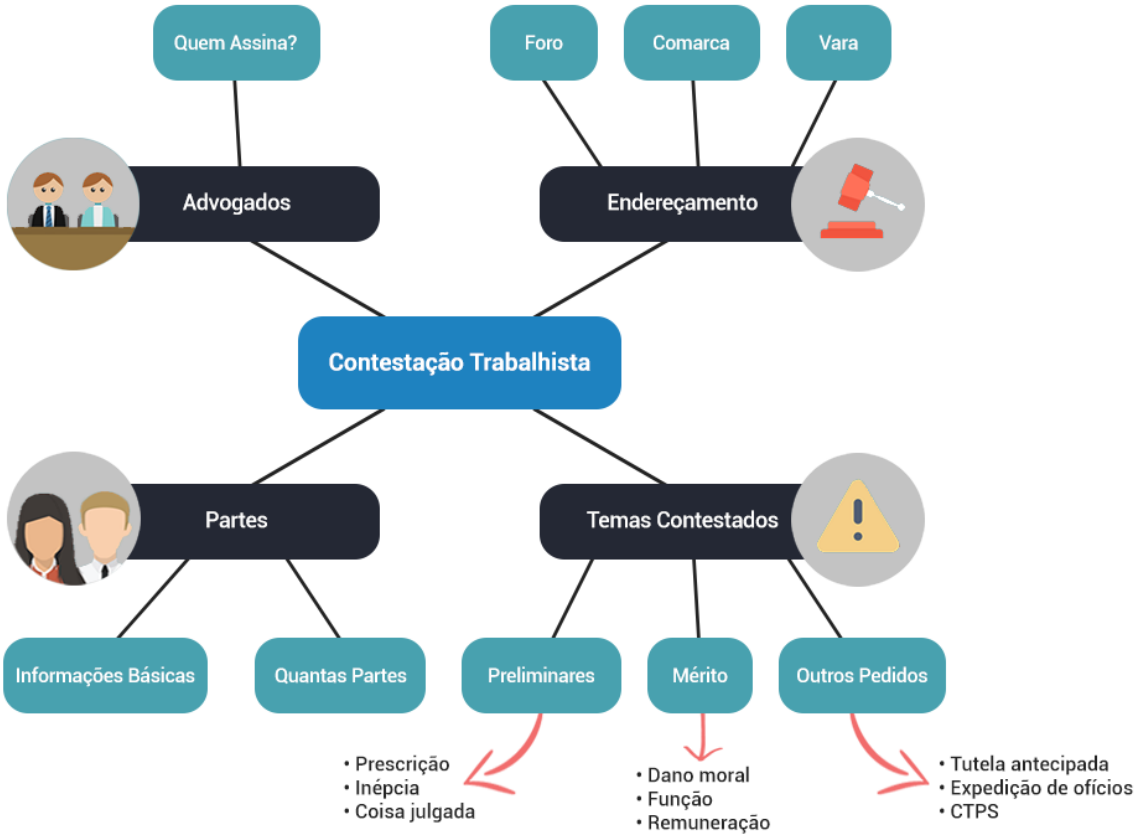

Figura 3.9: Etapa 4. Destacando subtemas e reforçando mensagens mnemônicas.

Diagrama entidade-relacionamento

É comum nos depararmos com o problema de identificar dentro um modelo de peça contratual, quais informações são variáveis entre exemplos diferentes. Quando pensamos em "nome completo", "RG", "CPF", "profissão", "estado civil" e "endereço residencial", logo nos vem à cabeça a "qualificação de uma pessoa natural capaz". Isso porque nosso cérebro já foi treinado o suficiente para representar grupos de informações de modo automático. Existem ferramentas que nos auxiliam a representar informações bem mais complexas que essas e relacioná-las entre si. Uma dessas ferramentas é o bem conhecido modelo **entidade-relacionamento**.

O **modelo entidade relacionamento** [15] - também chamado de modelo ER - é um modelo de representação composto por um conjunto de objetos básicos chamados de **entidades** e um conjunto de conexões entre esses objetos chamados de **relacionamentos**. Tal modelo também incorpora algumas informações semânticas básicas a respeito do mundo real. Embora o modelo ER seja amplamente difundido em disciplinas relacionadas com bancos de dados, essa ferramenta é especialmente útil para tarefas de automação jurídica, dado seu poder de representação de informações presentes em documentos jurídicos.

Uma **entidade** é um objeto que existe e é distinguível de outros objetos. Exemplos: *parte*, *advogado*, *contrato*, etc. Uma **classe de entidades** é um tipo de abstração usada para definir grupos de um mesmo objeto, ou seja, representa uma classe de coisas ou pessoas. Por exemplo, a classe de entidades *ser humano* representa um grupo de seres vivos dos quais todos nós fazemos parte. Daí surge o conceito de **instância** de uma classe de entidades. Uma instância é um exemplo, uma amostra específica, um elemento do conjunto daquela classe de entidades. No exemplo anterior, trata-se de uma instância da classe *ser humano*.

Cada entidade por sua vez é representada por um conjunto de **atributos**, a saber, propriedades que descrevem conjuntamente cada membro da respectiva classe de entidades. Em outras palavras, um atributo é um elemento informacional que contém o valor de uma propriedade de uma entidade. Por exemplo, todo *ser humano* possui propriedades como *nome*, *data de nascimento*, *idade*, *altura*, *peso*, *gênero*, etc. Tais propriedades são atributos da entidade *ser humano*. Observe que esses atributos são comuns à todas as instâncias da classe *ser humano*. Agora imagine o caso em que queremos modelar um atributo típico de um cidadão brasileiro: o *RG*. Será que todos os seres humanos possuem *RG*? Obviamente não. Visto que somente brasileiros são portadores de *RG*, tal atributo não é geral o suficiente para ser considerado atributo membro da entidade *ser humano*.

Os atributos por sua vez são definidos dentro de um domínio específico. Tal domínio é um conjunto de valores permitidos para cada atributo. Tais valores são por sua vez categorizados em tipos, dentre os quais se destacam os números, as datas, horas e os textos. Para se ter uma ideia de domínio, a idade de uma pessoa deve ser um número inteiro positivo. Não existem pessoas com idade negativa, por exemplo. O domínio de um atributo é, também, comumente definido como um conjunto de valores discretos, por exemplo, o gênero de um ser humano pode ser um dos valores {"F", "M"} do tipo texto. Outro aspecto interessante, é que podemos definir o valor de um atributo a partir de outros, como por exemplo, *idade* é um atributo que deriva de cálculos envolvendo o atributo *data de nascimento*. Dizemos nesses casos que o atributo *idade* é um atributo derivado.

Quanto a classificação de um atributo, seguem algumas definições. Um atributo *monovalorado* (ou atributo atômico) é aquele que admite somente um valor, por exemplo: uma pessoa tem apenas único valor textual para representar o atributo nome. Por outro lado, um atributo *multivalorado* (ou atributo não atômico) é aquele que capaz de admitir mais de um valor possível, por exemplo: uma pessoa pode possuir nenhum, um ou mais filhos. Atente-se ao atributo *filho*: observe que filho é também uma instância de *ser humano*. Isso pode? Sim. Um atributo pode também ser uma outra entidade composta por seus próprios atributos, como

no caso, o atributo *filho*. Quando ocorre algo assim, dizemos que existe um **aninhamento** de entidades por meio de atributos. O aninhamento por meio de um atributo define que tal atributo é composto, caso contrário é simples. Por exemplo, o atributo RG da entidade *pessoa natural brasileira* é também uma entidade, cujos atributos são *número*, *órgão expedidor* e *unidade federativa*. Os atributos por sua vez possuem seus respectivos atributos, como por exemplo, *número* é do tipo texto (sei que soa estranho, mas ele também admite pontos e traços). Para fins didáticos, a seguinte lista sumariza as classificações até aqui apresentadas:

- **Atributo simples**: não tem outros atributos aninhados, apenas o valor;
- **Atributo composto**: tem outros atributos aninhados;
- **Atributo monovalorado**: um único valor para cada instância e
- **Atributo multivalorado**: admite mais de um valor para cada entidade.

Pode-se assumir uma instância de entidade como o estado de uma entidade em um dado instante. O estado de uma entidade do conjunto é determinado pelos valores dos atributos dessa entidade. Quando se observa que em uma entidade existe um ou mais atributos que identificam unicamente cada instância de uma classe de entidades, esses atributos são chamados de atributos chave. Por exemplo, o *CPF* de uma *pessoa natural brasileira* é chave, já o atributo *nome* não é.

Conforme dito, as entidades são conectadas umas as outras através de relacionamentos. Um **relacionamento** é uma estrutura que indica uma associação entre instâncias de duas ou mais entidades. Quando um relacionamento une apenas duas entidades, dizemos que tal relacionamento é binário. Um relacionamento tem sempre um nome, normalmente relacionado ao verbo que estabelece a relação entre as entidades envolvidas. Por exemplo, o relacionamento *trabalhar* se conecta às entidades *empresa* e *empregado*. Além disso, é possível em alguns casos, definir atributos de um relacionamento. No exemplo anterior, o atributo *carga horária* pode ser considerado um atributo do relacionamento *trabalhar*, uma vez que é um atributo próprio da relação de trabalho, não da empresa ou do empregado. Assim como existe o conceito de classe de entidades como um conjunto de amostras daquela entidade, existe o conceito de classe de relacionamentos que descreve um conjunto de amostras (instâncias) do relacionamento em questão.

A cardinalidade de um relacionamento binário especifica a quantidade de membros que se relacionam entre uma parte e outra. No exemplo anterior, é possível que muitos empregados estabeleçam uma relação de trabalho com uma única empresa. Nesse caso, dizemos que se estabelece uma relação muitos para um entre empresas e empregados. Pode-se considerar nesse caso que uma empresa admite muitos empregados, mas um empregado só trabalha em uma única empresa, situação bem típica de acordo com o regime CLT. De um modo geral, as cardinalidades dos relacionamentos podem ser estabelecidas por relações do tipo:

- **Relação um-para-um**: ocorre quando apenas uma instância de uma entidade se relaciona com apenas uma instância de outra entidade. Exemplo A: uma *empresa* pode *possuir* somente um *contrato social*.
- **Relação um-para-muitos**: ocorre quando apenas uma instância de uma entidade se relaciona com múltiplas instâncias de outra entidade. Exemplo B: uma *empresa* pode *ter* vários *empregados*.
- **Relação muitos-para-muitos**: ocorre quando apenas múltiplas instâncias de uma entidade se relaciona com múltiplas instâncias de outra entidade. Exemplo C: muitas *empresas* podem *rescindir contratos* com muitos *empregados*. É comum confundir o relacionamento um-para-um com o relacionamento muitos-para-muitos. Para desambiguar esses casos, pense que muitos-para-muitos é equivalente a um-para-muitos e muitos-para-um concomitantemente. No exemplo C uma empresa pode rescindir

contrato com vários funcionários, assim como ao longo da vida, um funcionário pode rescindir contrato com várias empresas. O mesmo não se pode dizer do exemplo A, pois uma empresa não pode ter mais de um contrato social, nem um contrato social pode pertencer a mais de uma empresa.

Vale salientar que caso haja um relacionamento com mais de duas entidades, a cardinalidade se calcula da mesma forma, no entanto, com relações compostas (um-pra-muitos-para-um, um-pra-um-para-um-para-muitos, etc).

Após definir os conceitos fundamentais do modelo ER, vamos a sua representação gráfica. Seguem as definições das componentes gráficas do diagrama ER:

- **Classes de entidades**: são representadas por retângulos;
- **Atributos**: são normalmente representadas por elipses;
- **Classes de relacionamentos**: são normalmente representadas por losangos.
- **Linhas**: ligam atributos a classes de entidades e classes de entidades a classes de relacionamentos.

Todos os elementos acima são ligados por linhas, de modo que atributos podem ser ligados a outros atributos (multivalorados), a classes de entidades e classes de relacionamentos; no mais, entidades só se ligam a relacionamentos e vice-versa. É de costume indicar a direção do relacionamento por meio de linhas com setas. A cardinalidade do relacionamento é outro aspecto que se costuma denotar nas linhas.

Quanto aos atributos, se convenciona apenas descrever seus respectivos nomes. No entanto, vamos adotar uma variante na qual se especifica também o tipo do atributo à direita do mesmo. Atributos chaves serão nos nossos exemplos representados por elipses de linha dupla.

Observe na Figura 3.10 como ficaria um diagrama ER do último exemplo de relação de trabalho apresentado.

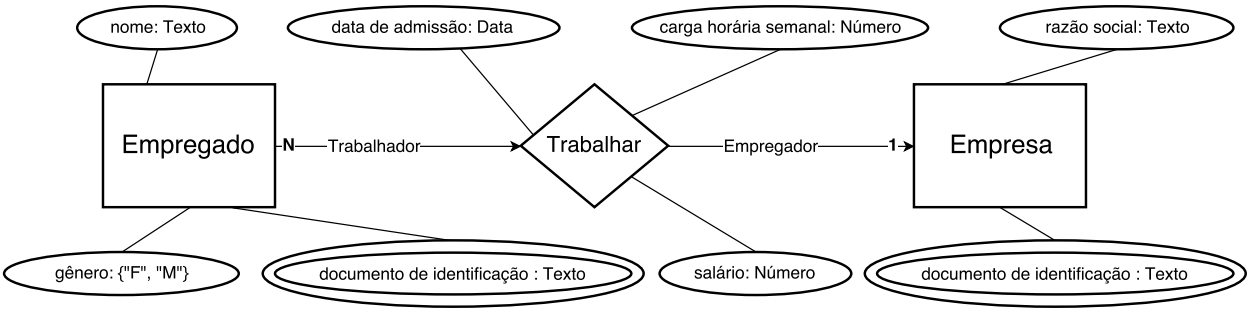

Figura 3.10: Diagrama ER para representar uma relação de trabalho básica.

A Figura 3.10 mostra como o *empregado* se relaciona com uma *empresa* diante do contrato de trabalho. A seta indica a relação ativa do trabalhador em relação a empresa. Se o verbo que descrever o relacionamento fosse "empregar", então as setas indicariam sentido inverso. Conforme definimos anteriormente, os atributos estão vinculados às classes de entidades e representadas pelas elipses. Observe que ao lado direito de cada atributo se especifica o tipo de cada um deles. Caso o tipo seja composto por uma lista valores, os discriminamos dentro de chaves. Outra observação importante é a notação do **papel** dentro do relacionamento sobre as linhas que unem as entidades aos relacionamentos. Tais notações enriquecem o diagrama e às vezes desambiguam múltiplas relações entre as entidades, como por exemplo, o papel do empregado na empresa

(administrador, projetista, etc). Por fim, observamos as cardinalidades associadas às entidades respectivas, denotadas próximas a ela sobre as linhas que se ligam ao relacionamento. Encontramos também o atributo *documento de identificação* definido como atributo chave para caracterizar tanto empregados, quanto empresas.

Vamos agora a um exemplo real no mundo jurídico. Para mostrar relações mais complexas entre entidades, optamos por modelar de um modo bem superficial um contrato de compra e venda entre duas pessoas. No caso do exemplo, considere que, por questões didáticas, todas as pessoas têm um *nome*, quer sejam pessoas físicas para representar o nome próprio, quer sejam pessoas jurídicas para representar a razão ou denominação social.

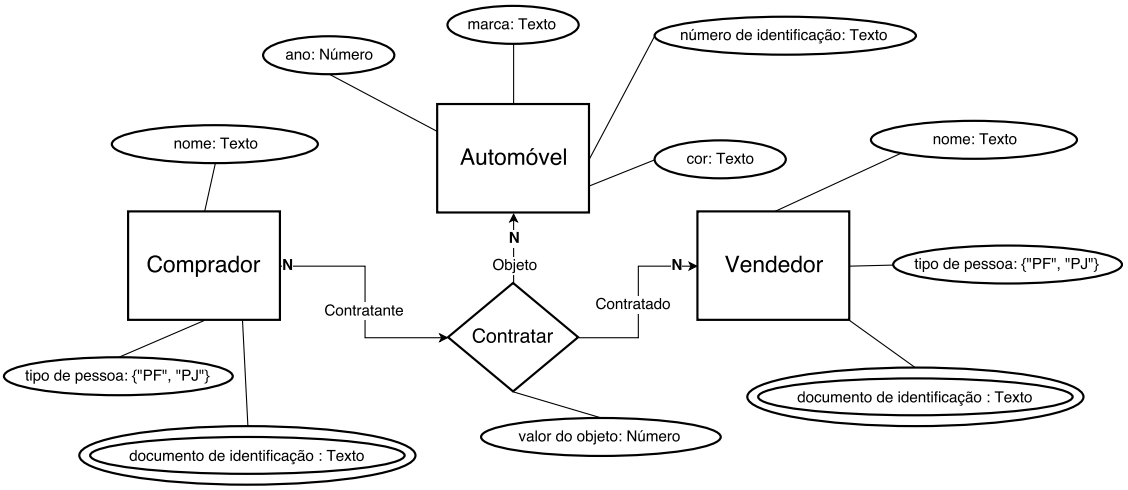

Figura 3.11: Visão geral de um contrato de compra e venda de automóveis.

Observe na Figura 3.11 uma visão superficial de como se relacionam um comprador e um vendedor diante de um contrato de compra e venda de automóveis. Obviamente, existem muitos desdobramentos que não foram considerados na diagramação. Assim mesmo, observe os detalhes do modelo ER tais como o tipo de relacionamento (ternário nesse caso), especificações dos atributos de cada classe de entidades, as definições dos tipos e domínios, cardinalidades, as setas evidenciando os polos ativo e passivo do acordo, o papel de cada um dentro da relação contratual, entre outros elementos. Ele traduz todo o conhecimento até aqui apresentado.

Uma técnica de diagramação, conhecida como **generalização**, consiste em encontrar classes de entidades comuns a duas outras subclasses pré-estabelecidas. No exemplo exposto na Figura 3.11, notamos que os atributos *nome*, *tipo de pessoa* e *documento de identificação* são atributos comuns a *vendedor* e *comprador*. Obviamente, isso ocorre pelo fato se que tanto compradores quanto vendedores são pessoas. Logo, podemos considerar a classe de entidades *pessoa* como uma generalização de comprador e vendedor.

Formalmente, a generalização é um processo *bottom-up* que combina classes de entidades que compartilham atributos entre si, em uma nova classe de entidades.

A Figura 3.12 mostra o mesmo exemplo da Figura 3.11, porém, usando a notação de generalização. Observe que a diagramação ficou muito mais limpa e clara que a anterior. Muitas vezes, no processo reverso a generalização é requerido, ou seja, as vezes é necessário estender uma classe de entidades muito genérica afim de adicionar novos atributos. Nesses casos, dizemos que ocorre uma *especialização* de classe de entidades.

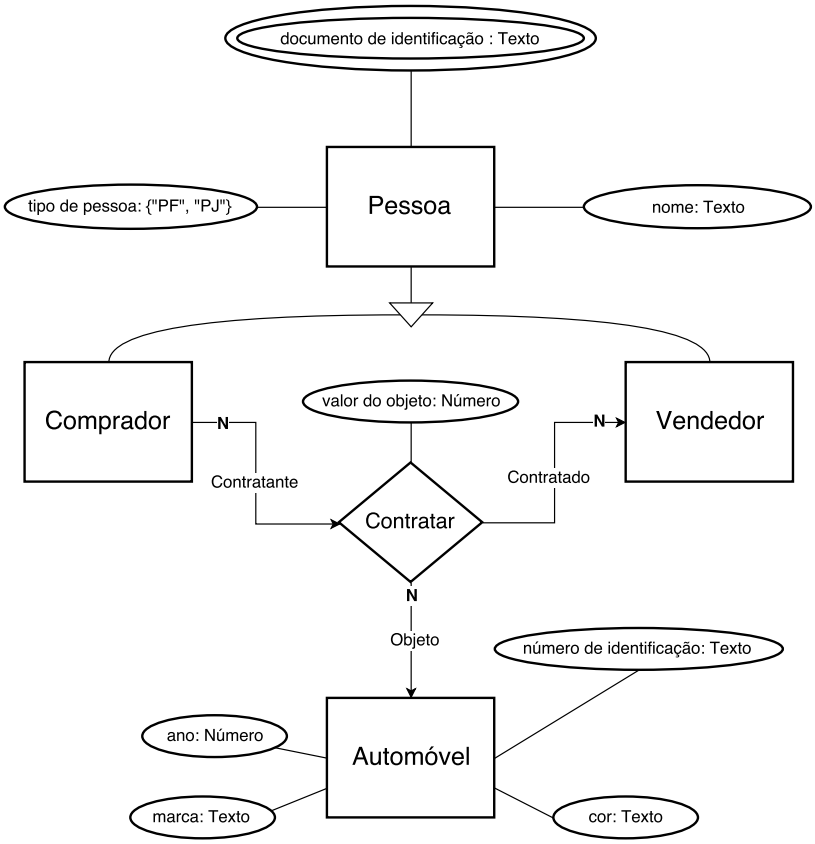

Figura 3.12: Visão geral de um contrato de compra e venda de automóveis usando generalização de pessoas.

Formalmente, a especialização é um processo *top-down* que designa uma subclasse de entidades dentro de uma classe de entidades pai que se distingue de outras. Especialização e generalização são simples inversões uma da outra, de modo que são igualmente representadas dentro do diagrama ER.

A especialização e a generalização definem um relacionamento conhecido na literatura como *herança*, uma vez que as subclasses, chamadas de classes filhas herdam todas os atributos da superclasse, as chamadas *classes pai*. Por exemplo: em biologia, a classe dos animais compreende as aves, anfíbios, mamíferos, répteis e peixes. Todas essas cinco subclasses compartilham propriedades comuns a todos os animais, como exemplo, todos eles são seres vivos que nascem, crescem, morrem e possuem capacidade de reprodução. Note que podemos especificar as classes até o nível de granularidade que interesse ao projeto, como por exemplo, especificar os mamíferos entre carnívoros, herbívoros e onívoros, ou ainda em aquáticos, terrestres e aéreos; tudo depende do critério de classificação, ou como bem se diz popularmente: “depende do ponto de vista”.

A componente gráfica que simboliza a generalização de entidades é um triângulo invertido.

Existem muitas maneiras e visões diferentes de modelar o mesmo problema. O leitor pode se questionar por exemplo se comprador e vendedor é de fato uma subclasse de pessoa ou simplesmente são funções que uma pessoa exerce diante de um contrato. É uma visão igualmente válida. Nesse caso, pode-se tomar o modelo ER exemplificado na Figura 3.12 e compactá-lo de modo que haja somente uma entidade pessoa. Neste caso, a classe de entidades *pessoa* passa a se relacionar não só com *automóvel*, mas também passa a se relacionar consigo mesma, o que caracteriza um **auto-relacionamento**.

A Figura 3.13 exibe essa visão de projeto. Basicamente, essa nova proposta de diagrama ER exclui as classes de entidades *comprador* e *vendedor* e as representa como uma única classe *pessoa*. As setas indicam o papel

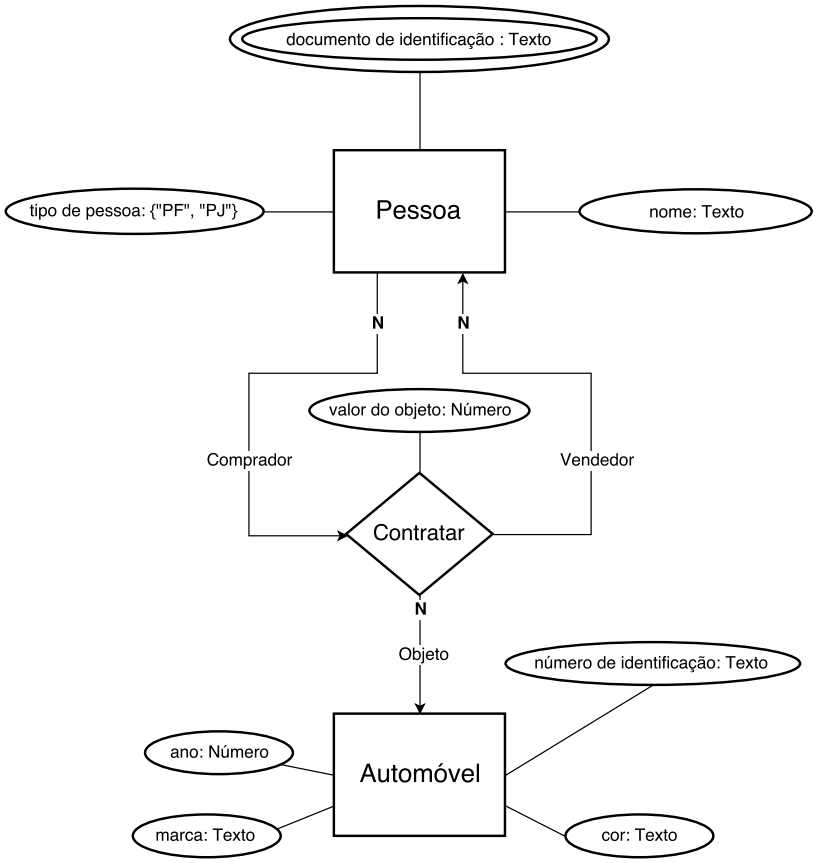

Figura 3.13: Visão geral compacta de um contrato de compra e venda de automóveis.

do comprador e do vendedor diante da relação contratual. Os demais dados permanecem inalterados em relação ao diagrama apresentado na Figura 3.12.

Conforme vimos, o modelo ER é uma ferramenta muito útil para extrair e organizar informações de um projeto a ser modelado. Um diagrama real completo pode ser tão grande quanto se queira detalhar o problema. Ele provê uma visão macro de como entidades e subentidades fundamentais se relacionam entre si. Note que o foco é a informação em si e não o fluxo dela dentro de um sistema. Existem outras ferramentas que também são úteis para definir relações entre entidades, tais como diagrama UML [21], onde além dos atributos de cada classe, também denota-se comportamentos que instâncias dessas classes possuem.

Exercício 2. Antes de prosseguirmos ao tema seguinte, pegue seu *mind map* elaborado no exercício anterior e eleve um nível de abstração: faça o modelo ER dele, exibindo as classes de entidades mais importantes, bem como seus relacionamentos intrínsecos. Não se esqueça de especificar bem os atributos e seus tipos, as cardinalidades e os papéis de cada entidade. Bom trabalho!

A seguir, apresentaremos uma ferramenta apropriada para definir fluxos informacionais que transitam no processo de automatização de documentos: a árvore de decisão.

Árvore de decisão e grafos

Antes de entender o que é uma árvore de decisão, vamos a alguns conceitos preliminares sobre **Teoria dos Grafos** [22]. A teoria dos grafos serve para modelar muitos problemas em vários ramos da matemática, informática, engenharia, indústria e muitas outras áreas. Para o Direito, ela é especialmente útil para modelar

relações entre objetos, cláusulas, sujeitos e qualquer elemento ou conceito que possam ser discretizado[12]. Ela favorece a visualização de esquemas, modelos, processos, algoritmos, e qualquer relação entre objetos que são combinados livremente.

Matematicamente, um **grafo** é um par (V, A) em que V é um conjunto arbitrário e A é um subconjunto de V. Os elementos de V são chamados vértices e os de A são chamados arestas. Um **vértice**, também chamado de **nó**, pode representar qualquer tipo de dado, processo, objeto, entidade ou qualquer outra coisa que se queira representar. No caso do diagrama ER por exemplo, entidades, atributos e relacionamentos são considerados vértices de um grafo. Uma **aresta** corresponde a conexão ou ligação entre dois vértices, digamos $\{a, b\}$. Nesse caso, a aresta $\{a, b\}$ será denotada simplesmente por ab ou por ba. Dizemos que a aresta ab incide em a e em b e que a e b são as pontas (ou extremidades) da aresta. Se ab é uma aresta, dizemos que os vértices a e b são vizinhos ou adjacentes. Além disso, quando a aresta indica uma direção (através de uma seta), dizemos que o grafo é **dirigido** e necessariamente $ab \neq ba$. Quanto à notação gráfica, os grafos são muito mais simples que o modelo ER por exemplo. Vértices podem ser representados por qualquer figura geométrica (quadrados, círculos, etc.) e as arestas podem ser qualquer tipo de linha (pontilhada, dupla, simples, etc.) que une dois vértices.

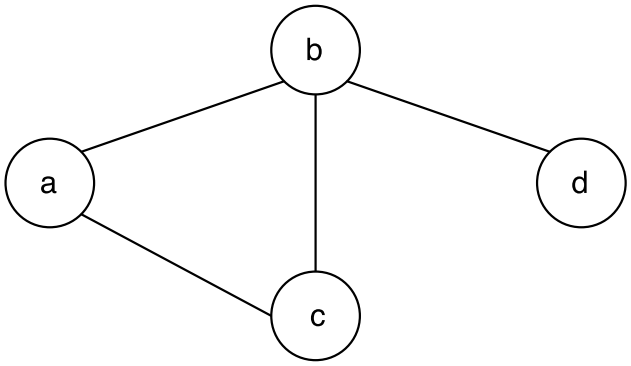

Figura 3.14: Exemplo de um grafo simples.

A Figura 3.14 apresenta um grafo com quatro vértices a, b, c e d e quatro vértices ab, ac e bc e bd. No caso, a e b são vizinhos, mas a e d não o são. Imagine que esse grafo pode representar uma rede social onde os vértices são pessoas e as arestas definem uma relação de amizade entre elas.

Quanto às arestas dos grafos, é comum atribuir pesos numéricos em muitos problemas combinatóriais. Sendo assim, qualquer aresta pode receber rótulos numéricos e não numéricos, dependendo da aplicação.

Numa aplicação bem simples dos conceitos adquiridos até aqui, imagine que uma família que vive em Belém do Pará e está em uma viagem de lazer, estando alojados na casa de um parente em Campinas. Como bons torcedores do Santos Futebol Clube, irão visitar o estádio Vila Belmiro a todo custo. Entretanto, cada integrante da família tem interesse em visitar uma das três cidades antes de ir para Santos: o filho mais velho quer conhecer uma moça que conheceu no WhatsApp em Sorocaba, a mãe quer passar no Brás para comprar roupas em São Paulo e a filha mais nova quer curtir os priminhos que moram em São José dos Campos. O pai de família, o seu Ari, analisando a situação financeira nem pestanejou: "Vamos realizar o trajeto que gaste o mínimo possível." Numa época em que não existia Waze ou Google Maps, o pai matemático pega uma folha de papel e desenha o grafo apresentado na Figura 3.15. Fazendo os cálculos, decidiu passar em São Paulo e deixar sua mulher mais contente que os demais. Neste caso, o seu Ari usou um grafo dirigido,

[12]Para fins do presente trabalho, a palavra **discretizar** deverá ser entendida como o ato de dividir ou fracionar um bloco grande de informações em pequenos blocos

com rótulos nas arestas para indicar a distância mínima entre as cidades. Uma sequência de nós que não se repetem dentro do grafo é chamada de **caminho**.

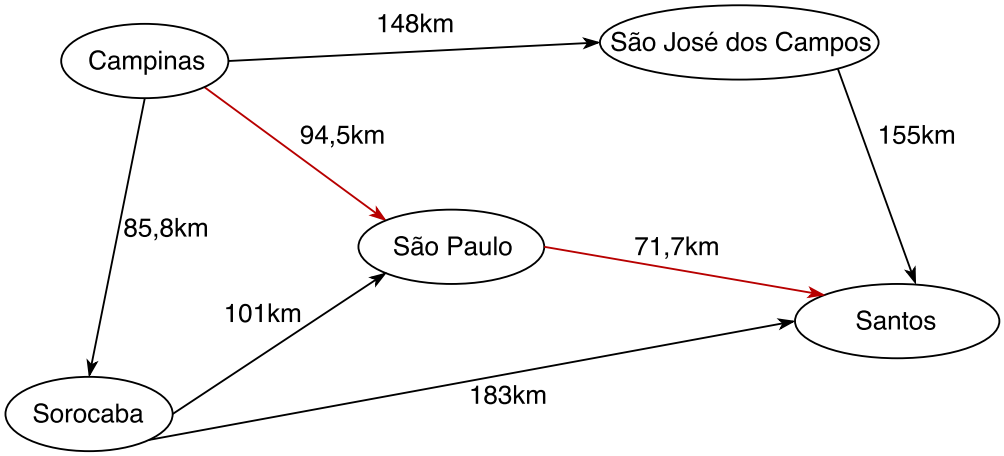

Figura 3.15: Grafo de planejamento de viagem do seu Ari.

Exercício 3. Que tal você fazer um grafo também? Nesse caso, imagine que você seja um distribuidor de cosméticos e precisa fazer entregas nos bairros: Sé, Brás, Perdizes, Jardim Paulistano, Moema, Morumbi e Tucuruvi. Seu tempo é curto e você está de motocicleta. Desconsiderando o trânsito, desenhe um grafo e decida qual a melhor rota para sua jornada.

O grafo é uma ferramenta muito usada para modelar, fazer prototipações e validar problemas abstratos do mundo real. Uma variedade inestimável de problemas pode ser modelado usando a Teoria de grafos. Muitos dos problemas sobre grafos tornaram-se célebres porque são um interessante desafio intelectual, bem como possuem importantes aplicações práticas, como por exemplo, o caminho mínimo, o caminho *Hamiltoniano*[13], ou o problema do caixeiro viajante[14], entre outros. No escopo do que esse trabalho propõe, optamos não entrar a fundo na exploração desses problemas, mas incentivamos que desbravem e amplifiquem seus horizontes através de pesquisas independentes. Na literatura existe material complementar aos montes [23, 24, 25, 26].

No mundo da inteligência artificial, as **árvores de decisão** [10] são modelos estatísticos que aprendem estruturas topológicas para a classificação e previsão de dados mediante uma etapa de treinamento supervisionada. As árvores de decisão são capazes de representar fluxos de dados e operações por meio de estruturas de seleção. Na teoria dos grafos, uma árvore é um grafo dirigido sem ocorrência de **ciclos**, ou seja, não existe caminhos circulares que iniciam em um vértice e termine nele próprio. O nó inicial é chamado de **raiz** da árvore, enquanto que os nós finais são chamados de **folhas**.

O raciocínio de se tomar uma decisão usando uma árvore de decisão é bem simples. Basta percorrer o fluxo de dados a partir da raiz da árvore (início) e ir percorrendo as arestas até encontrar uma folha (fim). Este método de classificação pode ser facilmente compreendido através de um exemplo.

Observe a árvore de decisão na Figura 3.16. Nesse exemplo vemos a raiz indicada pela estrutura de decisão

[13] O caminho *Hamiltoniano* é um problema combinatorial onde se deseja decidir se é possível percorrer por todos os vértices de um grafo uma única vez.
[14] Uma variante do caminho Hamiltoniano com custos associados às arestas.

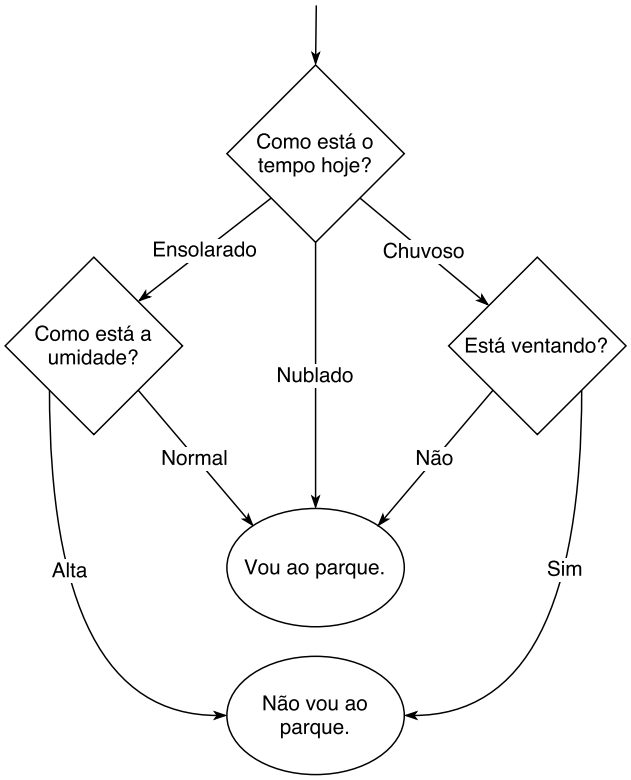

Figura 3.16: Exemplo de árvore de decisão bem simples.

"Como está o tempo hoje?". Caso o tempo esteja *nublado* a árvore direciona à folha "Vou ao parque". Além disso, se for um dia *ensolarado* mas com taxa de umidade considerada *normal*, ou se for um dia *chuvoso* e **sem ventos** também se decide ir ao parque. Caso contrário, isto é, se for um dia *ensolarado* mas com taxa de umidade *alta*, ou se for um dia *chuvoso* e *com ventos*, a árvore é direcionada à folha "Não vou ao parque".

Para a maioria das árvores de decisão o que vale é a tomada de decisão. Normalmente, todo o caminho percorrido para esse fim é descartado. Em casos de automação é bem diferente. O percurso dentro da árvore também é usado para gerar o documento. Nesse caso, o percurso realizado na árvore desde a raiz até uma folha dentro da árvore, definido como **caminho maximal**, contém todas as informações suficientes para gerar um documento completo. Para tanto, devemos definir em nossa árvore de decisão nós especiais para imprimir texto, realizar operações e repetições, em vez de simplesmente usá-lo como problema de classificação. Por essa razão, contrariando um pouco a definição de árvore segundo a teoria dos grafos, usaremos uma variação de árvore de decisão que admite ciclos para representar eventualmente estruturas de repetição. Note que cada caminho maximal da árvore de decisão corresponde a um documento concreto com informações estáticas a preencher, tal como ocorre em *malas diretas do Microsoft Word*.

Para fins de automatização de documentos, a árvore descreve uma sequência de regras que imprime texto e executa operações genéricas em uma certa ordem, seguindo o fluxo sequencial desde a raiz até uma folha, levando em consideração condições e repetições ao longo do caminho. Um nó também pode representar uma subárvore e assim progressivamente.

Vamos definir formalmente os elementos de uma árvore, conforme os seguintes componentes:

(I) **Requisição** é um componente usado para pedir informações ao usuário na árvore. Sua representação gráfica também é um **retângulo** e é visto como o **input** do sistema.

(II) **Texto** é um nó específico que imprime textos no documento final. Sua representação gráfica é uma

elipse e é visto como o **output** do sistema.

(III) **Operação** é um nó específico que realiza operações genéricas dentro do documento, tais como executar operações matemáticas, atribuição de valores, cálculos com datas, entre outras. Sua representação gráfica é um **trapézio**.

(IV) **Condição** é um nó que provê caminhos alternativos no fluxo de perguntas que direcionam a outros ramos da árvore de decisão. Sua representação gráfica é um **losango**.

(V) **Repetições**: caminhos da árvore que se repetem, podendo haver limitação quanto ao número mínimo e máximo de repetições. A repetição não possui um elemento simbólico gráfico como os demais elementos da árvore. Elas são representadas por desvios no fluxo descrito nas arestas até um ponto já visitado anteriormente na árvore.

(VI) **Tópico** é um nó especial que representa uma subárvore, ou seja, um *ramo* da árvore que agrupa regras sobre um mesmo tema. Dentro de um tópico podemos ter textos, condições, pedidos de resposta e até mesmo outros tópicos. Sua representação gráfica é um **hexágono**.

Vamos agora aplicar o conceito de árvore de decisão a um caso real do mundo jurídico. Vamos fazer uma procuração *ad judicia et extra*. A árvore de decisão resultante é apresentada na Figura 3.17. Acompanhe por alguns minutos os possíveis caminhos dirigidos pelas arestas a partir da raiz. Esse acompanhamento guiado pelas setas determina o fluxo de execução das instruções contidas na árvore de decisão. A primeira parte se destina à qualificação do outorgante e do outorgado. Em seguida pergunta-se se a parte outorgada é um advogado. Se sim, há uma especificação a respeito do tipo de procuração *ad judicia* (com ou sem processo em curso) ou *et extra*, do contrário já pede informações sobre os poderes outorgados. Ainda no primeiro caso, note que se já existe processo em curso, não se especifica os poderes outorgados. Na requisição e impressão dos poderes outorgados se observa uma estrutura de repetição, uma vez que o número de poderes outorgados não é determinado. Tal estrutura é indicada pela aresta de retorno. Após a impressão dos poderes outorgados se pede informações sobre poderes de substabelecimento. Caso seja permitido substabelecer, a árvore indaga sobre reserva de poderes e imprime os textos adequados e pergunta na sequência sobre o prazo. Caso não se opte por estabelecer poderes, pergunta-se diretamente sobre os prazos. No caso dos prazos, ou se define prazo determinado ou considera-se o prazo indeterminado. Obviamente ao se escolher prazo determinado deve-se perguntar qual é o prazo e imprimí-lo na sequência. Finalmente a árvore pede os dados de local e data da assinatura para subsequente impressão.

Exercício 4. Vamos exercitar a compreensão da árvore de decisão. De acordo com a árvore do exemplo da Figura 3.17, descreva o que se imprimirá nas seguintes situações:

1. Quero uma procuração *et extra* sem substabelecimento para meu advogado sem prazo de expiração.
2. Quero renovar uma procuração dada a um advogado por mais 6 meses, visto que tramita um processo anterior representado por ele. Essa procuração continuará podendo ser subestabelecida com reservas de poderes.

É provado que as árvores de decisão são capazes de modelar um volume gigante de documentos sem ser pela linguagem computacional. Note que com uma árvore de decisão, conseguimos guardar um número exponencial de possibilidades de se compor um documento no caso concreto. Em outras palavras, uma árvore

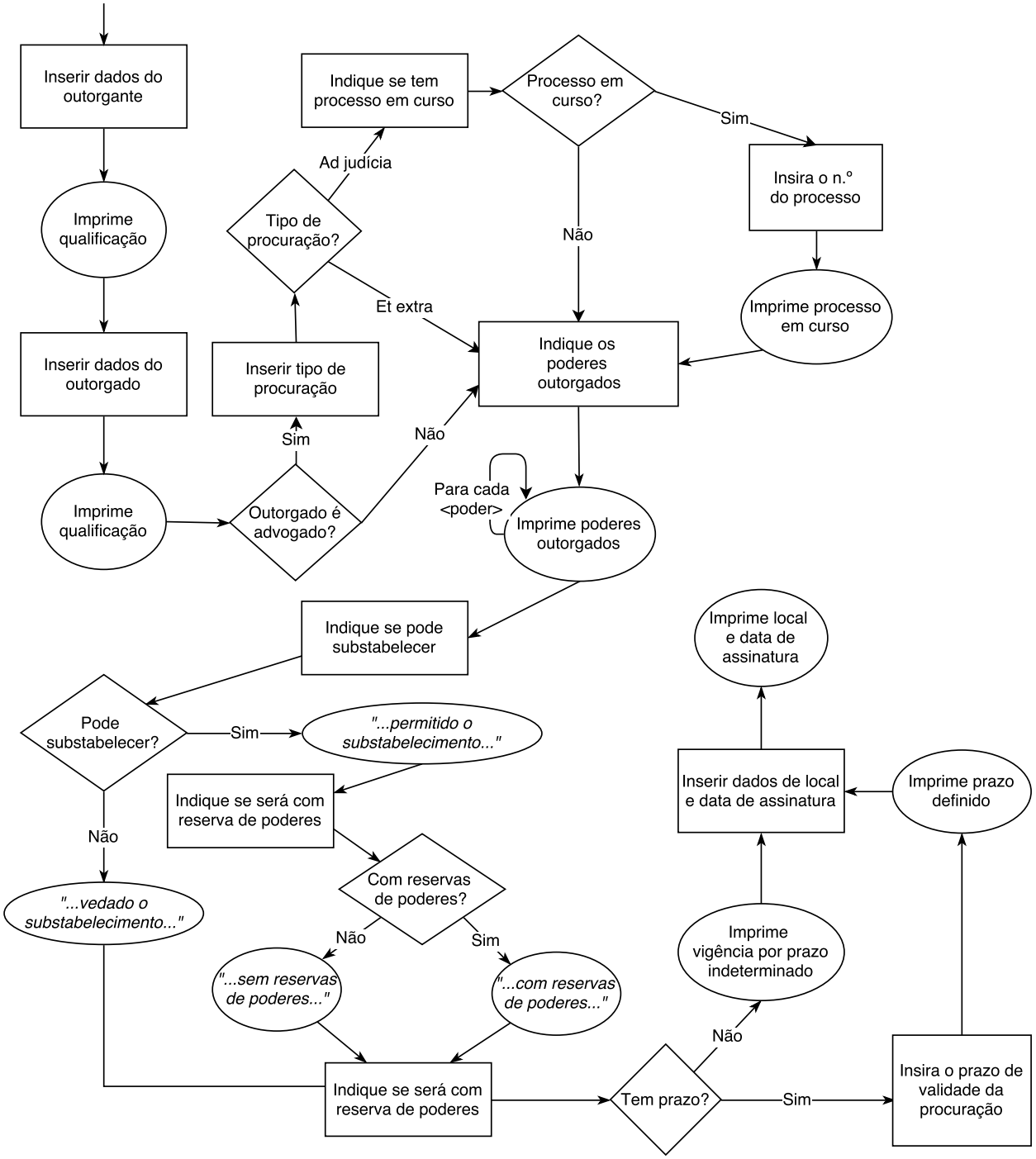

Figura 3.17: Exemplo de árvore de decisão de uma procuração.

de decisão representa um número gigante de permutações e possibilidades de composição de documentos jurídicos. Cada caminho numa árvore de decisão é uma das permutações por ela compreendida. É comum vermos árvores de permutações como ilustrações em introduções à análise combinatória. Por outro lado, essas representações são prontamente abandonadas após as introduções, pois as representações visuais não comportam a complexidade de alguns problemas. Esboçar visualmente uma lógica que você quer inserir no código pode fazer sentido, mas a capacidade de abstrair é importante também. No início é comum usarmos ferramentas como o `http://draw.io/` para esboçar as árvores ou grafos, mas ao longo do tempo acabamos por abandonar essa etapa e representar as coisas diretamente em código e como no filme Matrix, começamos a entender as telas cheias de letras caindo. Entretanto, não se perca! Alguns cuidados devem ser tomados.

Métodos de elaboração - Top-down *versus* *Bottom-up:* Agora, como fazer uma árvore de decisão? Existem duas formas basicamente: Ou você escolhe um monte de **casos concretos** de documentos a se automatizar e vai derivando a árvore progressivamente a partir desses casos; ou você gasta um tempo raciocinando detalhadamente, imaginando a partir de **casos abstratos**, quais as entidades que compõem a árvore e vai montando como um quebra-cabeças. Essa tarefa costuma determinar se um **projetista** tem aptidão para ser engenheiro ou arquiteto jurídico.

Definimos o método de elaboração por composição (método **bottom-up**) como a elaboração da árvore a partir de técnicas de engenharia reversa, ou seja, a partir de *casos concretos*. Através da análise de um ou mais documentos, se estrutura as teses e o texto já consolidado e explícito, definindo os tópicos e fragmentos que sofrerão alterações dependendo de cada condição. Nessa abordagem de "baixo para cima", os tópicos básicos são inicialmente descritos em detalhes. Então, esses tópicos são associados de modo a compor um tópico maior. Esse tópico maior por sua vez, pode ser associado a outros tópicos maiores ainda, e assim, progressivamente, vai compondo outros níveis superiores até que se chegue ao nível mais elevado do documento: a árvore de decisão final.

O procedimento básico de uma metodologia *bottom-up* segue aproximadamente o seguinte padrão:

1. ***Topificação***: Dado um conjunto considerável de amostras de documentos que compõem a base de conhecimento jurídico para um modelo específico, recorte fragmentos do documento e classifique-os por assuntos ou tópicos.

   Por exemplo, ao tomar-se variadas amostras de contratos em geral, entre outros tópicos encontramos: Objeto do Contrato; Prazo; Qualificação das Partes; Valores e Forma de Pagamento; Disposições gerais; Considerandos e Rescisão. *Brainstormings* são fundamentais para auxiliar as tomadas de decisão. Do mesmo modo, o modelo entidade-relacionamento se mostra bastante apropriado para representar os tópicos fundamentais do modelo.

2. **Alinhamento**: Após ter em mãos os tópicos principais de vários documentos analisados, se combina as permutações possíveis a fim de caracterizar um nível mais elevado da árvore de decisão. Na fase de alinhamento, classifica-se trechos de documentos de acordo com a topologia dos tópicos anotados. No exemplo anterior, se percebeu que alguns temas se repetem com frequência, tais como:

   - Considerandos;
   - Qualificação das Partes;
   - Valores e Forma de Pagamento;
   - Prazo;
   - Disposições gerais;
   - Rescisão; e
   - Objeto do Contrato.

   Conforme feito no exemplo aqui apresentado, a fase de alinhamento consiste em agrupar os trechos de documentos em **grupos topológicos**, que são partes de documentos topologicamente idênticas em documentos diferentes. A utilidade de se agrupar é ter uma delimitação mais precisa do escopo do trabalho. É como perceber que existem alguns trechos dentro de documentos que ocorrem muito raramente. Talvez isso indique que tal trecho não merece tanta atenção em uma primeira versão do

modelo. Em outras palavras, grupos topológicos com poucas amostras podem ser desconsideradas a princípio.

Após se agrupar os fragmentos em grupos topológicos, um passo intermediário antes de se compor a árvore de decisão, é elevar o nível topológico de modo a compreender a topologia "macro" do documento como um todo. Assim, conseguimos entender a ordem em que os tópicos aparecem no texto, se são condicionados ou repetidos dentro da árvore. No caso do exemplo anterior, constatamos que os temas estão dispostos em uma ordem preferencial:

Tópico 1: Qualificação das Partes;

Tópico 2: Considerandos;

Tópico 3: Objeto do Contrato;

Tópico 4: Prazo;

Tópico 5: Valores e Forma de Pagamento;

Tópico 6: Rescisão; e

Tópico 7: Disposições gerais.

3. **Composição da árvore**: Este processo é realizado a princípio dentro de cada grupo topológico "relevante" a fim de criar pequenas árvores para cada grupo topológico, e progressivamente, ir combinando as árvores entre si. Se tomássemos uma sequência contínua de tópicos encontrados dentro de um documento qualquer e criássemos uma árvore de decisão, poderíamos compor um caminho da árvore de decisão (segundo o conceito de grafo). Assim, existe uma equivalência entre uma sequência de tópicos e um segmento de um caminho (um ramo) na árvore de decisão.

   Tecnicamente, podemos conceituar a composição da árvore da seguinte forma: Considere que $P(g_i)$ seja o **caminho** dentro do grupo topológico g equivalente ao i-ésimo trecho de documento catalogado dentro do grupo. O processo de montagem da árvore escolhe inicialmente um par de caminhos quaisquer, digamos $P(g_1)$ e $P(g_2)$, e os combina de modo a formar uma árvore inicial $T_{(1,2)}$. A partir dessa árvore $T_{(1,2)}$ se escolhe um novo caminho candidato $P(g_3)$ a fim de compor uma nova versão da árvore, digamos a árvore $T_{(1,2,3)}$. E assim, progressivamente até que se tenha uma árvore final $T_{(1,2,3,\ldots,n)}$ resultante, oriunda da combinação de todos os caminhos catalogados. Esse processo é chamado de *processo iterativo de composição da árvore de decisão*. Note que o processo se inicia do nível mais concreto (o texto) até um nível mais abstrato (a árvore).

   O processo iterativo de composição combina dois caminhos diferentes, duas árvores ou um caminho e uma árvore de decisão levando em consideração: (a) os tópicos que estão repetidos nos dois caminhos, ou seja, são idênticos e possuem o mesmo nome; (b) a ocorrência de tópicos em apenas um dos caminhos devido à existência de estruturas condicionantes; e (c) a ocorrência de tópicos redundantes dentro do próprio caminho ou da árvore, caracterizando assim, a presença de estruturas de repetição. A Figura 3.18 exemplifica cada um dos casos descritos, devidamente indicadas. Os símbolos $\oplus$ e $\Rightarrow$ significam "combinação" e "conclusão", respectivamente.

   Com uma árvore pré-estabelecida, conseguimos abreviar bastante o escopo do projeto, de modo a ser possível recombinar ou mesmo recompor a redação de texto proposta pelos casos concretos.

4. **Definição dos textos**: Após ter sido alinhado topologicamente os trechos comuns aos documentos, o passo a seguir é a composição da redação final que será impressa dentro de cada tópico. Observe

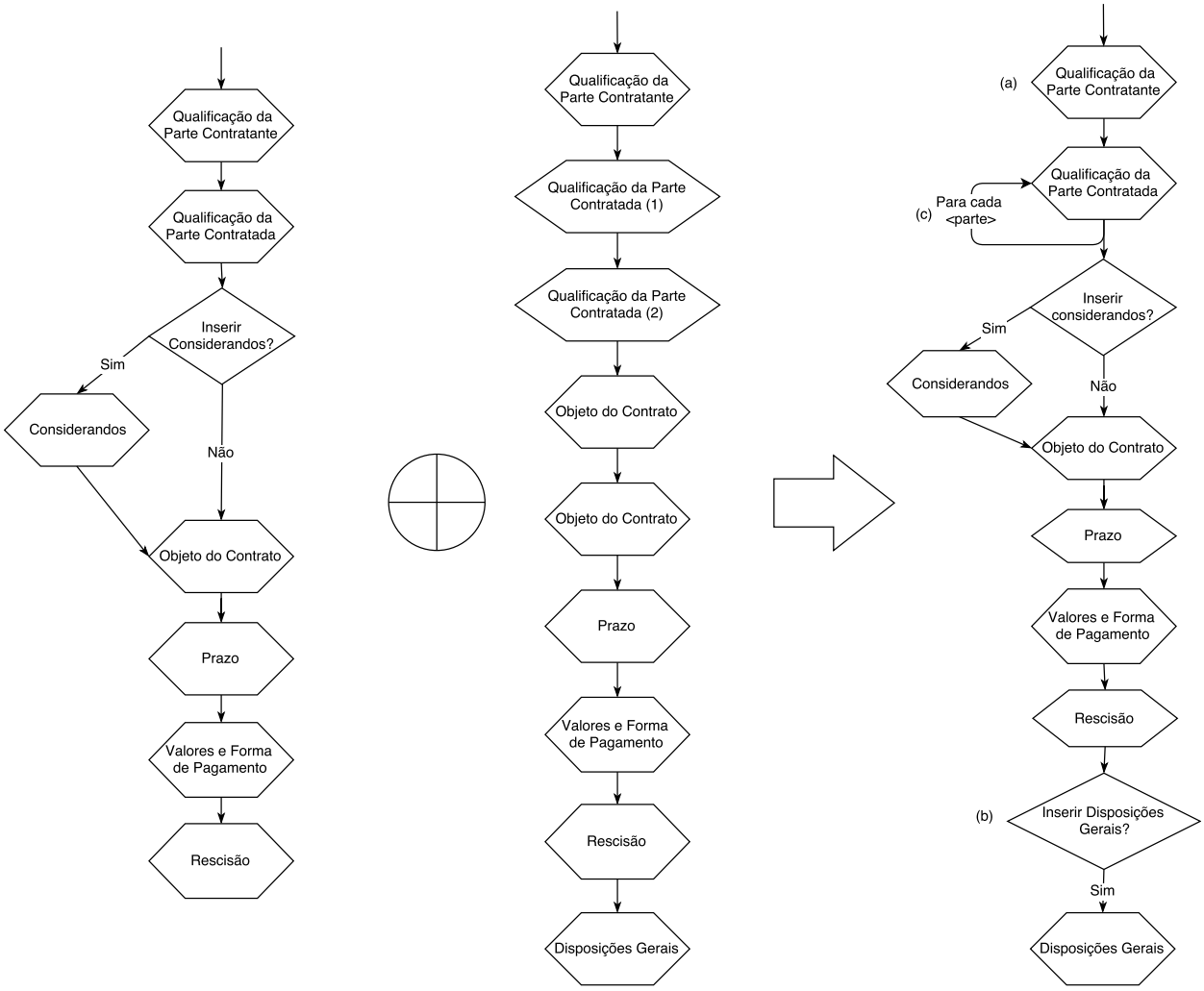

Figura 3.18: Operações de composição possíveis: (a) colapsamento de nós repetidos, (b) desdobramento de caminhos por desvio condicional, e (c) compactação por estruturas de repetição.

que esse procedimento poderia ter sido feito logo após a categorização topológica. Entretanto, como vimos que existem tópicos redundantes em grupos diferentes, isto é, texto repetitivo, optamos poupar trabalho e realizar essa etapa após a eliminação de redundâncias ter sido executada.

O objetivo principal dessa fase é encontrar uma versão final para cada impressão e inserí-las na árvore de decisão. Embora não seja um trabalho tão complexo, a pluralidade de escritas às vezes nos deixa em dúvidas sobre qual a melhor redação. Na dúvida, siga a redação que mais vezes aparecem no texto, ou melhor, reescreva sua versão melhorada delas.

Uma outra etapa importante na tarefa de definição de texto é a indicação de termos que variam entre um trecho textual e outro equivalente em termos topológicos. Observe que textos equivalentes de documentos topologicamente idênticos possuem informações que variam entre si. Por exemplo, os dados das partes, datas, valores e demais informações que são específicas de uma amostra concreta. Essas informações são chamadas de **variáveis** e denotadas por <**nome da variável**>. Quando se pode agrupar variáveis em torno de um contexto, podemos definir grupo de variáveis, denotadas por <**:nome do grupo:**>.

O acesso a uma variável interna a um grupo de variáveis pode ser realizada mediante ao uso do ponto

('.') como de separador, por exemplo, um grupo de dados de uma corporação, digamos <:corporação:>, que possui informações como nome fantasia, CNPJ, razão social, podem ser acessadas, respectivamente, como <corporação.nome_fantasia>, <corporação.CNPJ> e <corporação.razão_social>.

O **valor** de uma variável corresponde ao dado que ela armazena. É toda informação que ela guarda com fins de executar expressões, comparações, impressões de texto etc. O valor de uma variável pode vir de uma resposta dada ao usuário do modelo ou pode ser alimentada internamente pelo projetista a partir de uma atribuição. Considere o exemplo de uma cláusula de contrato de investimento a seguir:

> As Partes acordam que, com contagem a partir 05/03/2018, data de contratação do Outorgado como prestador de serviços da Companhia ("Período Inicial"), após o decurso do período de 1 (um) ano, o Outorgado ficará *vested em* ou seja terá o direito de exercer a opção de compra sobre, 2.000 (duas mil) quotas, sem valor nominal da Companhia, sendo que a cada período de 1 (um) ano o Outorgado ficará *vested em* um adicional de 1.500 (mil e quinhentas) quotas, sem valor nominal, da Companhia, de forma que após decorridos 3 (três) anos do Período Inicial, findo em 05/03/2021 ("Período de Vesting"), o Outorgado terá o direito de exercer a compra da Participação Total, nos termos do presente Contrato.

Observe que existem alguns termos que costumam variar com frequência entre contratos diferentes. Esses termos são as variáveis. Note que fica a cláusula reescrita com termos que comumente variam segundo um critério arbitrário:

> As Partes acordam que, com contagem a partir <data de contratação>, data de contratação do Outorgado como prestador de serviços da Companhia ("Período Inicial"), após o decurso do período de 1 (um) ano, o Outorgado ficará *vested em*, ou seja terá o direito de exercer a opção de compra sobre, <número de quotas> (<número de quotas por extenso>) quotas, sem valor nominal, da Companhia, sendo que a cada período de 1 (um) ano o Outorgado ficará *vested em* um adicional de <número de quotas adicionais> (<número de quotas adicionais por extenso>) quotas, sem valor nominal, da Companhia, de forma que após decorridos <período de vesting> (<período de vesting por extenso>) anos do Período Inicial, findo em <data do fim do vesting> ("Período de Vesting"), o Outorgado terá o direito de exercer a compra da Participação Total, nos termos do presente Contrato.

Após essa etapa, passa-se a integrar os textos finais à árvore de decisão.

5. **Validação**: Por fim, é essencial realizar uma validação *bottom-up* fina a fim de comparar dados de documentos reais contra a árvore produzida. O processo se inicia da raiz e o validador responde as informações inseridas e percorre os caminhos verificando a completude e consistência da árvore.

A técnica de composição *bottom-up* pode ser aplicada não somente a documentos inteiros, como também a cláusulas específicas.

Vamos à aplicação em um exemplo bem simples: qualificação de pessoa. Considere que tenhamos os seguintes exemplos:

EXEMPLO (1) "Aline Pereira da Silva, brasileiro, solteiro, arquiteta, portadora do documento de identidade RG número 2342332-9, e inscrito no Cadastro de Pessoa Física sob o número 231.223.608-08, domiciliada na Rua José Paulo da Fonseca, nº 2321, São Paulo, SP, por meio de seu advogado, que esta subscreve"

EXEMPLO (2) "Empresa X3PO, inscrita no Cadastro Nacional de Pessoa Jurídica (C.N.P.J.) sob o número 76.177.354/0001-78, inscrição estadual sob o número 110.042.223.114, sediada na Rua Manoel dos Anjos, nº 231, 1º andar, Bairro Juraci, Jericoacoara - PE, de direito privado, por meio de seu advogado e bastante procurador, que esta subscreve"

EXEMPLO (3) "Brutus Quintus, grego, divorciado, portador do documento de identidade RG número 3253241-2, e inscrito no Cadastro de Pessoa Física sob o número 323.212.324-29, domiciliado na Rua Castro Alves, nº 53A, apto 32, Sorocaba, SP, por meio de seu advogado, que esta subscreve"

EXEMPLO (4) "RMKG Solutions, inscrita no Cadastro Nacional de Pessoa Jurídica sob o número 23.534.253/0001-01, sediada na Avenida Rui Barbosa, nº 452, Campo Grande, Mato Grosso do Sul, de direito privado, por meio de seu advogado e procurador, que esta subscreve"

Didaticamente, vamos aplicar os passos descritos anteriormente a esses casos.

1. ***Topificação***: Em uma granularidade bem fina, podemos enxergar cada termo do texto como se fossem tópicos. Como encontramos os seguintes temas dentro dos exemplos, os tópicos são definidos como: nomes, nacionalidade, estado civil, profissão, RG, CPF, endereço, texto de subscrição, razão social, CNPJ, inscrição estadual e texto de especificação de direito privado. Quanto à ordem desses tópicos, prevaleceu a seguinte sequência: nome/razão social, nacionalidade, estado civil, profissão, RG, CPF/CNPJ, endereço e texto de subscrição.

   Endereço por sua vez, se pode encontrar os subtemas: logradouro, número, complemento, bairro, cidade e UF.

2. **Alinhamento**: Observa-se a presença de uma variável oculta que determina se o sujeito qualificado é uma pessoa física ou jurídica.

   Quanto aos grupos topológicos, temos trechos que se especializam e descrevem o *endereço*, trechos que qualificam *pessoas físicas* e trechos que qualificam *pessoas jurídicas*.

3. **Composição da árvore**: Acompanhe o processo de composição da árvore de decisão, que dados poucos exemplos, optamos por realizar o processo em paralelo. Decompomos a árvore final em três **sub-árvores** representadas pelos grandes blocos em cinza.

   Observe na Figura 3.19 que usamos apenas três caminhos para montar a árvore de decisão do endereço. Isso porque a qualificação de endereço dos Exemplos 1 e 4 são redundantes. A Figura 3.20 descreve uma árvore de decisão para pessoas físicas e a Figura 3.21 para pessoas jurídicas. Nos três casos, realizamos combinações repeitando as regras de colapsamento de tópicos, desdobramento de caminhos e definição de repetição (embora não fosse encontrada nenhuma).

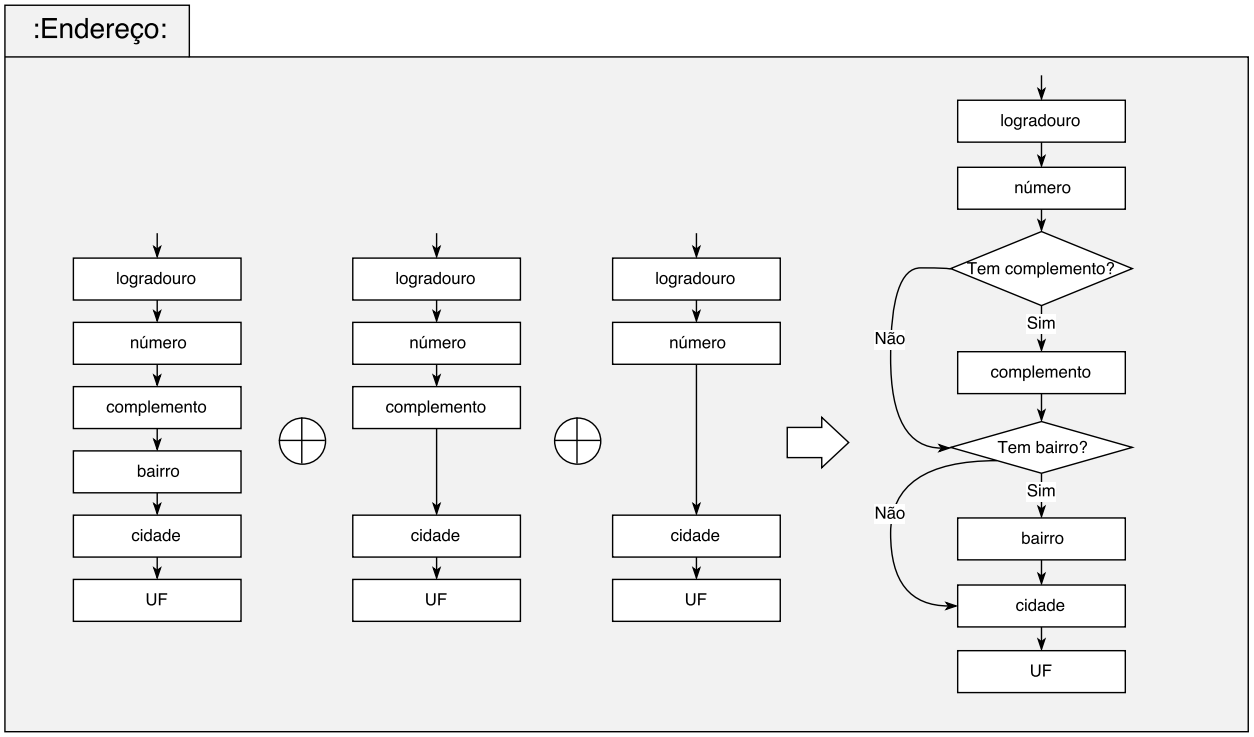

Figura 3.19: Composição iterativa da árvore de decisão para "endereço".

Por convenção, usaremos sub-árvores definidas anteriormente a partir de sua indicação delimitada por dois pontos (':'). Assim, você consegue manter a árvore limpa e fazer referências cruzadas.

Um aspecto interessante de compor a árvore de decisão em sub-árvores independentes é a capacidade de reaproveitamento de componentes, uma propriedade conhecida como **modularização**.

4. **Definição dos textos**: Como foram encontrados três grupos topológicos, conseguimos estipular um texto para cada um deles:

> **GRUPO (1) - endereço:** "<logradouro>, nº <número>, [complemento,][bairro,]<cidade>, <UF>"

> **GRUPO (2) - Pessoa Física:** "<nome>, <nacionalidade>, <estado civil>, [<profissão>,]portador(a) do documento de identidade RG número <RG>, e inscrito(a) no Cadastro de Pessoa Física sob o número <CPF>, domiciliado(a) na <:endereço:>, por meio de seu advogado, que esta subscreve"

> **GRUPO (3) - Pessoa Jurídica:** "<razão social>, inscrita no Cadastro Nacional de Pessoa Jurídica (C.N.P.J.) sob o número <CNPJ>, [inscrição estadual sob o número <inscrição estadual>,]sediada na <:endereço:>, de direito privado, por meio de seu advogado e bastante procurador, que esta subscreve"

Os textos escritos dentro de colchetes indicam texto opcional a depender da existência da informação dentro contida. Os nomes delimitados por '<' e '>' são informações que variam dentro do texto, dados do usuário.

5. **Validação**: Agora basta validar a árvore de decisão final, conforme é apresentada na Figura 3.23. O

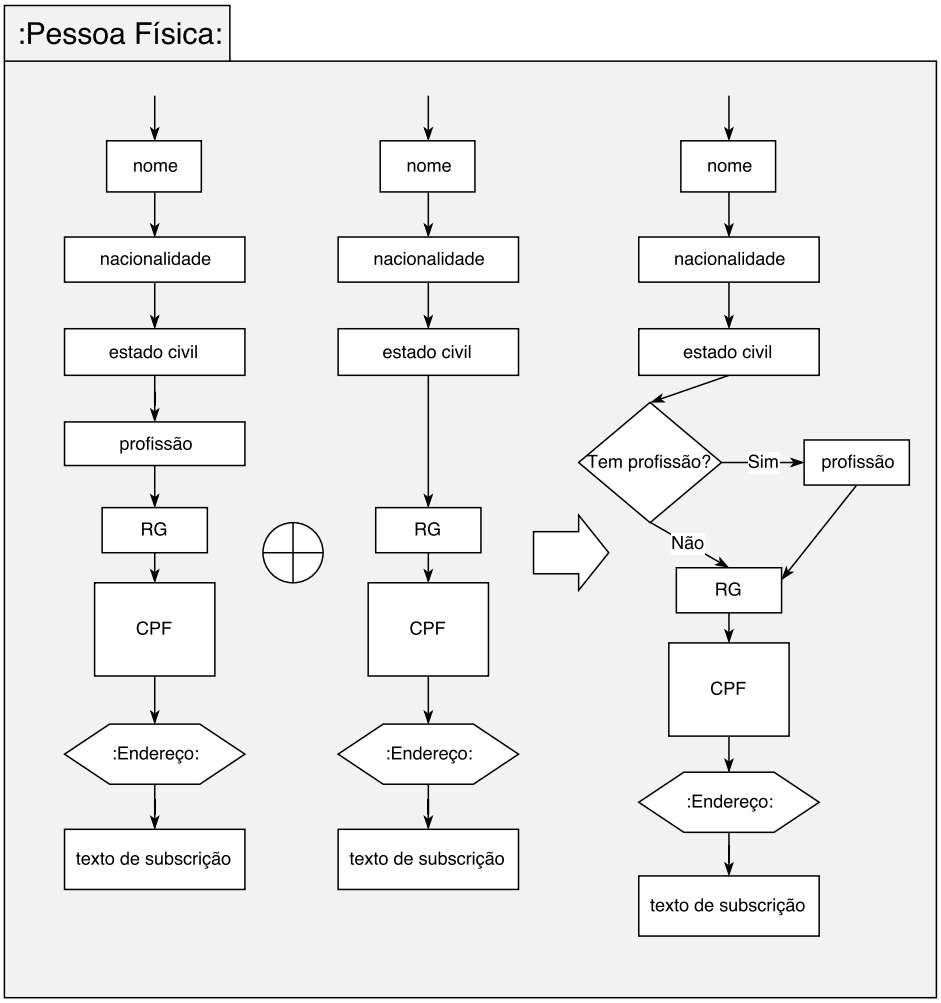

Figura 3.20: Composição iterativa da árvore de decisão para "Pessoa Física".

teste é feito simulando respostas na árvore de decisão (nos retângulos que representam entradas de dados) e ir escrevendo os textos dentro de cada elipse (que representa saída de texto).

Exercício 5. Agora é sua vez: Selecione dez exemplos de contratos de locação básicos e elabore uma árvore decisão usando a abordagem *bottom-up*.

Definimos por outro lado, o método de elaboração por decomposição (método **top-down**) como a elaboração de árvores a partir de técnicas de engenharia direta, isto é, a partir de *casos abstratos*. Através de casos abstratos construídos na mente do projetista, a árvore de decisões é montada aos poucos desde a superfície do modelo. Então, se pensa quais as ramificações mais importantes que um modelo pode ter. Sendo assim, o esboço do projeto vai se concretizando aos poucos. A medida que a estrutura do documento vai ficando mais clara, a definição da ordem das perguntas e o texto final flui naturalmente. Cada nível vai sendo detalhado, do mais alto ao mais baixo nível, de forma a se chegar nas especificações dos níveis mais básicos de cada elemento da árvore.

O procedimento básico de uma metodologia *top-down* segue aproximadamente o seguinte padrão:

1. **Conceituação**: Quais informações eu preciso para elaborar uma árvore de decisão, ou seja, quais

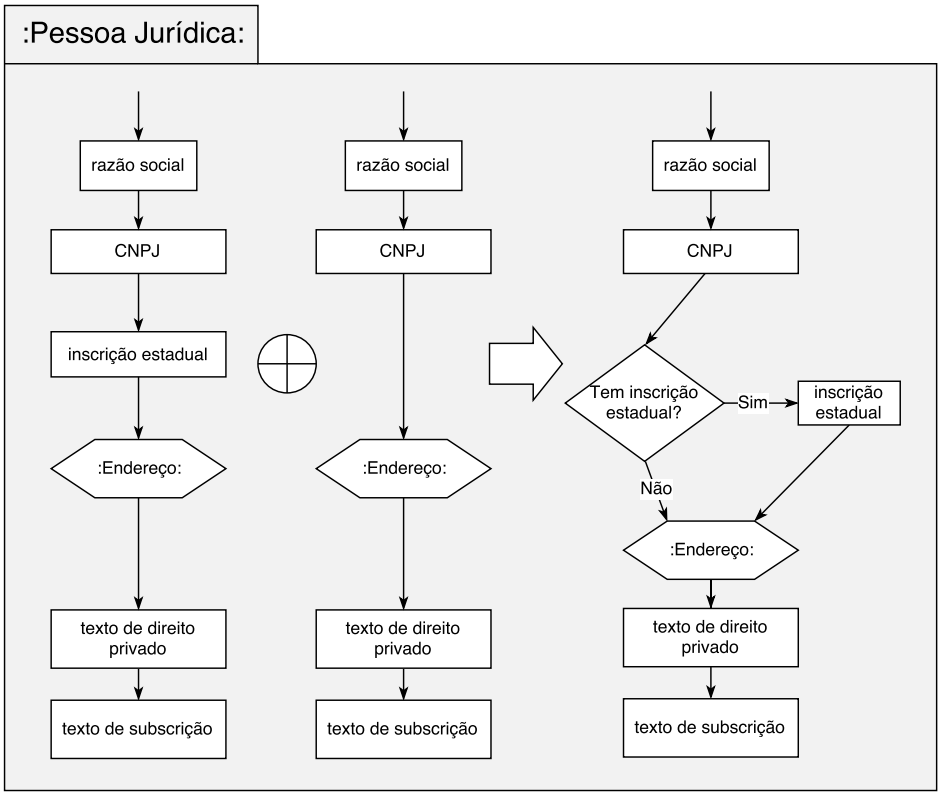

Figura 3.21: Composição iterativa da árvore de decisão para "Pessoa Jurídica".

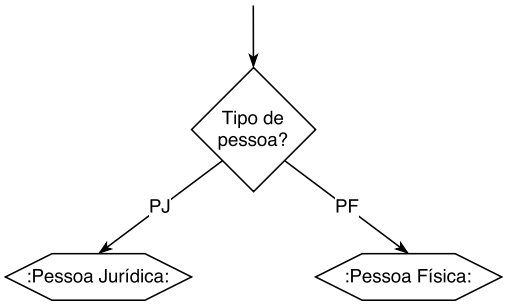

Figura 3.22: Resultado final da árvore de decisão da qualificação.

entidades eu preciso considerar na hora de estabelecer a topologia[15] do documento? Ferramentas como o **mind mapping** e o modelo entidade-relacionamento são muito apropriados para detectar quais as entidades fundamentais do modelo. Por exemplo, para fazer um contrato de locação, é necessário, no mínimo, de: dados da parte locadora, dados da parte locatária, do imóvel, de eventuais garantias e outros dados do contrato que podem variar (seguros, benfeitorias, responsabilidades etc.)

2. **Montagem da árvore**: Após a conceituação, é hora de ligar os pontos da cabeça das árvores de decisão. Nesse caso, não existe mistério, comece pelas entidades macro definidas na fase de conceituação e a medida que vão surgindo novos tópicos, vá agregando-os independentemente na árvore final. É um processo de construção e reconstrução constante, onde caminhos surgem, se dividem (no caos de estruturas condicionantes) e desaparecem a todo momento. Os caminhos tendem a esticar na medida em que a árvore atinge um estado de maturidade final. Mas tenha cuidado e não se perca em suas inócuas elucubrações. Elas podem destruir todo o projeto. Antes de começar, avalie e limite o escopo do projeto para que atenda a maioria (não todos) dos casos. O princípio de *Pareto*[16] deve ser

[15] Topologia de um documento pode ser entendida como a disposição, a forma e a organização das componentes primordiais que compõem a árvore de decisão.

[16] Regra dos 80/20 que diz que: para muitos eventos, aproximadamente 80% dos efeitos vêm de 20% das causas.

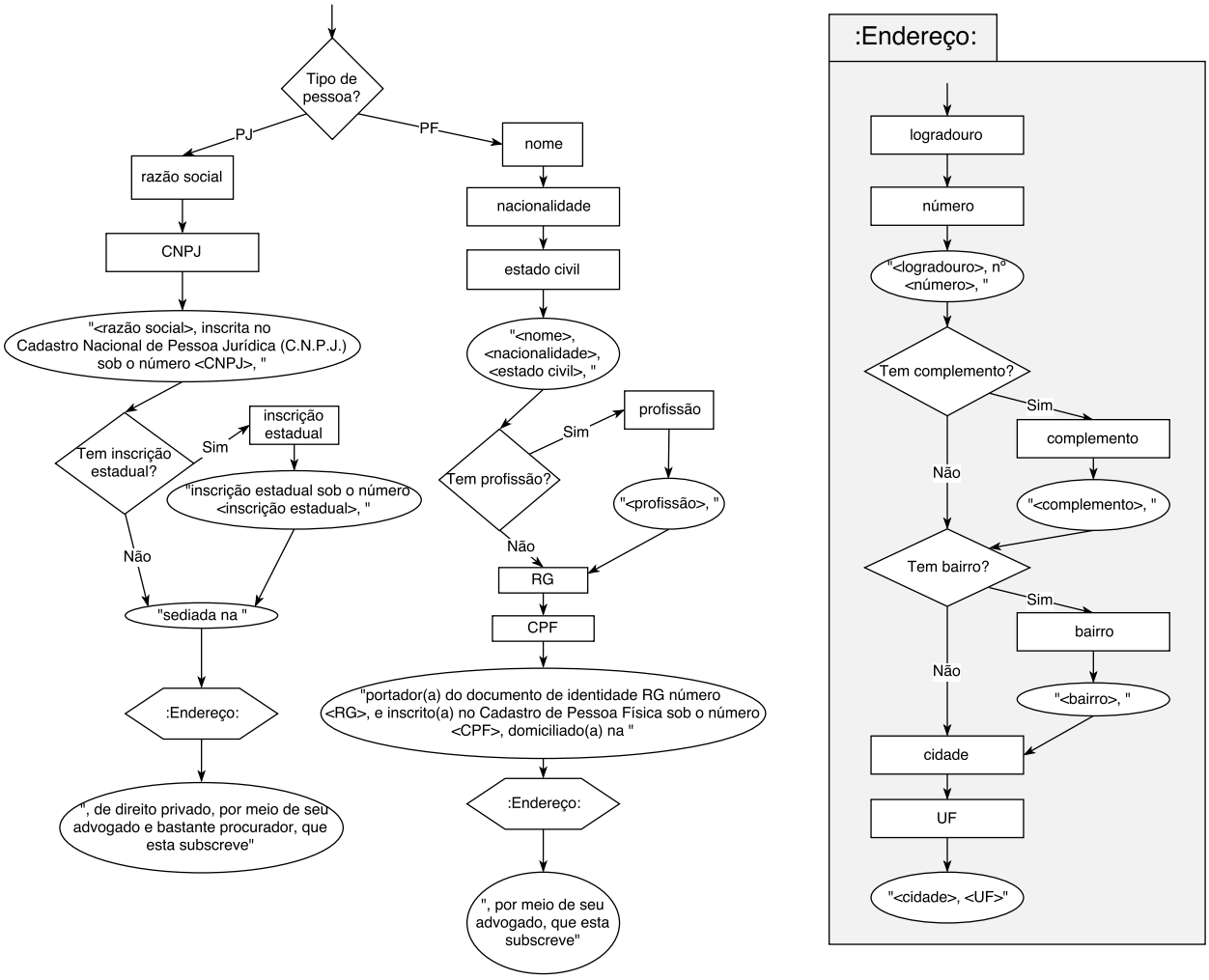

Figura 3.23: Resultado final da árvore de decisão da qualificação com textos.

internalizado nessa tarefa.

3. **Decomposição dos tópicos**: Como a árvore já em uma versão considerada estável, o próximo passo é a decomposição de cada tópico em uma subárvore. Por exemplo, dentro do tópico "Capital Social" de um contrato social podem existir desdobramentos infindáveis como "Tabela de participação societária", "Cláusulas de integralização total" ou "Cláusulas de integralização parcial" do capital, "Divisibilidade das quotas", "Vedação a oneração", "Cessão e transferência de quotas", etc. Perceba que o próprio tópico "Capital Social" é uma árvore de decisão por si só. Nesse caso, o que deve ser feito é voltar ao passo (1) e conceituar novamente o tópico para que posteriormente seja montada a árvore de acordo com o passo (2). O processo se dará por encerrado após constatada a completude do tópico a um nível satisfatório. Esse processo é chamado de *processo iterativo de decomposição da árvore de decisão.*

4. **Elaboração dos textos**: Após ter sido esgotado o processo iterativo de composição da árvore, é o momento de escrever as cláusulas. Diferentemente do método *bottom-up*, não existe nenhum texto de referência para essa fase. Pode-se extrair texto de minutas seletas, de exemplos pré-selecionados, ou compô-los da experiência adquirida ao longo dos anos. Uma vantagem grande do método *top-down* em relação ao anterior é a diminuição do efeito "colcha de retalhos" dos documentos produzidos, uma vez que a linguagem usada, o estilo de redação, os termos usados na escrita vêm de uma mesma pessoa. A cabeça do projetista normalmente concentra o *know-how* suficiente para elaborar as teses nesse tipo de abordagem.

5. **Validação**: Analogamente ao processo apresentado no método *bottom-up*, o processo de validação realiza testes reais de modo a garantir a validade, completude e consistência da árvore final.

Um exemplo sucinto da metodologia *top-down*, vamos montar a árvore de decisão que representa um modelo de contrato de locação de imóvel residencial urbano. Na fase de **Conceituação**, delimitado um escopo bem simples, chegamos a sete cláusulas fundamentais:

1. *Qualificação das partes.* Quem são as partes que têm a intenção de assinar o contrato?
2. *Objeto do contrato.* No caso da locação de um imóvel residencial urbano.
3. *Prazos do aluguel.* Essencial num contrato de aluguel.
4. *Data de pagamento e valor do aluguel.* Igualmente indispensável num contrato de aluguel.
5. *Obrigações do locatário.* De um modo mais geral, cobrir apenas casos mais clássicos.
6. *Penalidades.* Aplicadas ao locatário por descumprimento de alguma cláusula.
7. *Formas de rescisão e término do contrato.* Observar legislação no que diz respeito à locação por temporada.

Os tópicos abordados são facilmente revelados em reuniões de *brainstorming* e comparações de *mind maps.* de pessoas diferentes. Passando à etapa de **Montagem da árvore**, é razoável considerar a topologia dessa árvore inicial como um caminho natural: Qualificação das partes $\Longrightarrow$ Objeto do contrato $\Longrightarrow$ Valor do aluguel $\Longrightarrow$ Data de pagamento do aluguel $\Longrightarrow$ Obrigações do locatário $\Longrightarrow$ Penalidades $\Longrightarrow$ Formas de rescisão e término do contrato.

Conforme preconiza o método *top-down*, os próximos passos são aplicações recursivas (recorrentes) da re-conceituação e re-montagem da árvore. Assim, mergulhamos em mais um nível de refinamento dos tópicos encontrados, de modo que passamos a ter uma visão mais profunda a respeito de cada tema.

1. *Qualificação das partes.* Informações básicas mas importantes como nome completo, estado civil, profissão, RG, CPF, além do papel dentro do contrato (locador, locatário e fiador).
2. *Objeto do contrato.* Descrição do imóvel tais como o número da matrícula no Registro de Imóveis e número de contribuinte do IPTU, endereço completo e metragem.
3. *Prazos do aluguel.* Considere em um contrato convencional de período mensal. Além da data de entrada do locatário deve-se indicar se o contrato é por tempo determinado ou indeterminado.
4. *Data de pagamento e valor do aluguel.* A informação sobre o dia do pagamento do aluguel é necessária. Deve-se levar em conta também as despesas que serão de responsabilidade do locatário, como por exemplo o valor do condomínio.
5. *Obrigações do locatário.* Proibições e obrigações específicas: silêncio após às 22h, manter áreas públicas limpas, regras de visitantes, compromisso com o regulamento interno, além de indicar serviços não cobertos pelo aluguel como consumo de água, luz e telefone, entre outras.
6. *Penalidades.* Em particular, penalidades em casos de depredação do imóvel. Inclui-se também multas sobre atraso de pagamento, e a figura solidária do fiador, caso aplicável.
7. *Formas de rescisão e término do contrato.* Cláusulas de rescisão por prazos determinados ou por falta de pagamento.

O processo poderia ir ganhando complexidade progressivamente, por exemplo, com a consideração de outros tópicos importantes que aparecem em certos contratos de locação, tais como, tratamento específico em caso de reformas ou benfeitorias, cláusulas gerais de endereçamento e disposições gerais, bem como a previsão de vendas com o contrato de locação em vigor. Todos esses temas, por ocorrerem em casos específicos, geram desdobramentos na árvore de decisão final, os quais são modelados com estruturas de decisão.

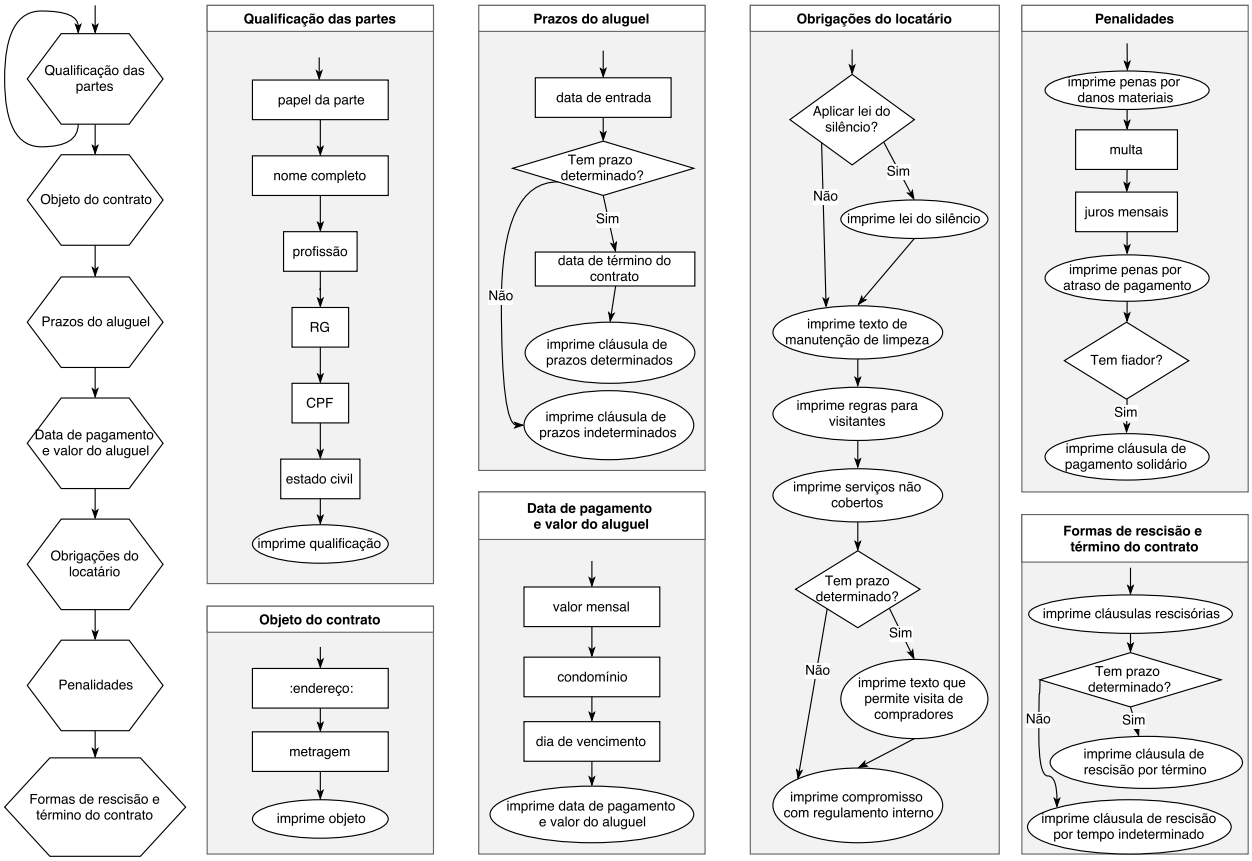

Figura 3.24: Árvore de decisão de um contrato de locação residencial elaborada com o método *top-down*

A Figura 3.24 apresenta uma proposta de árvore de decisão básica para contratos de locação residencial, baseado nos tópicos levantados mentalmente. Após ter sido definido um esboço relativamente estável da árvore, se inicia então a fase de composição do texto: Nesse caso, cada cláusula de impressão é associada a um nó de impressão da árvore, como proposto no exemplo abaixo:

1. **qualificação**: <papel da parte>: <nome completo>, brasileiro, <estado civil>, <profissão>, portador da cédula de identidade R.G. nº<RG> e CPF/MF nº <CPF>.

2. **objeto**: O objeto deste contrato de locação é o imóvel residencial com <metragem>m^2, situado à <endereço.logradouro>, <endereço.número>, <endereço.bairro>, CEP <endereço.CEP>, <endereço.cidade>, <endereço.estado>.

3. **cláusula de prazos determinados**: O prazo da locação inicia-se em <data de entrada> com término em <data de término do contrato>, independentemente e aviso, notificação ou interpelação judicial ou mesmo extrajudicial.

4. **cláusula de prazos indeterminados**: O prazo da locação inicia-se em <data de entrada> presumidamente por prazo indeterminado, sendo acordado entre as partes com pelo menos com um mês de antecedência.

5. **data de pagamento e valor do aluguel**: O aluguel mensal, deverá ser pago até o dia <dia do vencimento> do mês subsequente ao vencido, é de R$ <valor mensal> mensais, reajustados anualmente, de conformidade com a variação do IGP-M apurada no ano anterior, e na sua falta, por outro índice criado pelo Governo Federal, reajustamento este sempre incidente e calculado sobre o último aluguel pago no último mês do ano anterior. O LOCATÁRIO será responsável por todos os tributos incidentes sobre o imóvel bem como despesas ordinárias de condomínio, avaliado na presente data a R$ <condomínio>, reajustado conforme deliberações acordadas em assembleias regulares.

6. **lei do silêncio**: O proprietário ou o possuidor de um prédio tem o direito de fazer cessar as interferências prejudiciais à segurança, ao sossego e à saúde dos que o habitam, provocadas pela utilização de propriedade vizinha.

7. **texto de manutenção de limpeza**: Fica ao LOCATÁRIO, a responsabilidade em zelar pela conservação, limpeza do imóvel, efetuando as reformas necessárias para sua manutenção sendo que os gastos e pagamentos decorrentes da mesma, correrão por conta do mesmo.

8. **regras para visitantes**: O LOCATÁRIO declara, que o imóvel ora locado, destina-se única e exclusivamente para o seu uso residencial e de sua família. A visita a terceiros deverá ser comunicada à portaria em qualquer situação.

9. **serviços não cobertos**: O LOCATÁRIO deverá arcar com as despesas provenientes de sua utilização seja elas, ligação e consumo de luz, força, água e gás que serão pagas diretamente às empresas concessionárias dos referidos serviços.

10. **texto que permite visita de compradores**: Após comunicação por escrito, ao LOCADOR é permitido a realização de visitas comerciais guiadas, a fim de expor o imóvel objeto desse contrato a potenciais compradores, conquanto que seja marcada com pelo menos 1 (uma) semana de antecedência.

11. **compromisso com regulamento interno**: O LOCATÁRIO, obriga por si e sua família, a cumprir e a fazer cumprir integralmente as disposições legais sobre o Condomínio, a sua Convenção e o seu Regulamento Interno.

12. **penas por danos materiais**: O LOCATÁRIO obriga-se a manter o imóvel locado em perfeito estado de conservação e asseio, mantendo sempre em condições de higiene e limpeza, com os acessórios, partes hidráulicas e elétricas em perfeito estado de funcionamento, com pintura recente como declara neste ato receber, conforme laudo de vistoria em anexo, comprometendo-se ainda a fazer às suas expensas, sem ônus atual ou futuro para o LOCADOR, a pintura do imóvel e os reparos imediatos originários do uso, quer seja pela má conservação, quer seja pelo desgaste ou pelo abandono.

13. **penas por atraso de pagamento**: Em caso de mora no pagamento do aluguel, será aplicada multa de <multa>% sobre o valor devido e juros mensais de <juros mensais>% do montante devido.

14. **cláusula de pagamento solidário**: O FIADOR é o principal pagador do LOCATÁRIO, respondendo solidariamente por todos os pagamentos descritos neste contrato bem como, não só até o final de seu prazo, como mesmo depois, até a efetiva entrega das chaves ao LOCADOR e termo de vistoria do imóvel.

15. **cláusula de rescisão por término**: Finda a presente locação no dia <data de término do contrato>, o LOCADOR mandará proceder vistoria, após comunicação por escrito o LOCATÁRIO, a fim de verificar se o imóvel está nas mesmas condições em que foi locado, conforme laudo, ficando o LOCATÁRIO obrigado às indenizações pelos estragos que forem constatados, inclusive a renovação geral da pintura do imóvel, com utilização das tintas nas cores e marcas definidas pelo LOCADOR.

16. **cláusula de rescisão por tempo indeterminado**: Finda ou rescindida a presente locação, o LOCADOR mandará proceder vistoria, após comunicação por escrito o LOCATÁRIO, a fim de verificar se o imóvel está nas mesmas condições em que foi locado, conforme laudo, ficando o LOCATÁRIO obrigado às indenizações pelos estragos que forem constatados, inclusive a renovação geral da pintura do imóvel, com utilização das tintas nas cores e marcas definidas pelo LOCADOR.

A prática de organizar textos e descrevê-los dentro de tabelas ajuda a desacoplar os elementos de texto das árvores de decisão, e mantê-las limpas. Tais tabelas são denominadas **tabelas de definição de prints**.

Uma vez mais vale salientar que o exemplo é meramente ilustrativo e não comporta as nuances de um projeto completo de automação. Após encerrar os processos de criação do modelo, inicia-se os testes para validação da árvore. Neste caso, é importante tomar um grupo de documentos reais a fim de se verificar, especialmente, a completude dos documentos derivados do modelo.

Exercício 6. Partindo do contrato de aluguel apresentado, escolha um modelo de documento jurídico de mesma complexidade e construa uma árvore decisão usando a abordagem *top-down*.

Em suma, uma árvore de decisão pode ser produzida do abstrato para o concreto (*top-down*) ou do concreto para o abstrato (*bottom-up*). Cada problema demanda uma solução específica e é de se esperar que o melhor método combina diferentes abordagens. Evidenciou-se que o método *top-down* exige um conhecimento mais profundo do direito, com uma visão abstrata mais aguçada do modelo a ser automatizado, enquanto que o método *bottom-up* necessita de um volume considerável de amostras para mapear permutações relevantes de um modelo. Observe ainda que existe uma associação direta entre as competências de um arquiteto jurídico e o método *top-down*, bem como existe uma associação direta entre as competências de um engenheiro jurídico e o método *bottom-up*.

Um dos problemas comuns em projetos de árvore de decisão, é facilidade de se perder num escopo muito mais amplo do que o esperado. Esse escopo pode se dar tanto na especificação exagerada de um documento (*overfitting effect*) ou na generalização exagerada de muitos documentos (*over smoothing effect*). Classifica-se uma árvore de decisão segundo sua abrangência de duas formas possíveis:

- Por um lado, uma árvore de decisão é dita profunda (*deep tree*) quando seu escopo é mais limitado em número de permutações (poucas estruturas de decisão) mas com muitos nós especializados principalmente em operações e impressões de texto. No limite, temos uma árvore que faz poucos tipos de documentos, mas bem especialistas.
- Por um lado, uma árvore de decisão é dita larga (*wide tree*) quando seu escopo é mais abrangente em número de permutações (muitas estruturas de decisão) mas com poucos nós especializados em operações e impressões de texto. No limite, temos uma árvore que faz muitos tipos de documentos, mas bem superficiais.

Uma árvore pode ser tão grande quanto se deseja. No entanto, não dispomos de tempo infinito para realizar tudo que desejamos. Sendo assim, é necessário limitar o escopo e desenvolver o projeto em ondas de atualização de modo equilibrado e consciente. Há quem opte por construir árvores que possam gerar poucos documentos mais completos e progressivamente adicionar mais casos. Há quem opte por cobrir mais casos,

ainda que com um texto mais genérico. A decisão cabe ao projetista, também conhecido como arquiteto jurídico.

Mapas declarativos: Um mapa declarativo é uma estrutura de dados usada para catalogar as meta-informações de variáveis e operações mais complexas. Esses mapas são normalmente elaborados por arquitetos e repassados para que os engenheiros codifiquem.

Ao detectar termos variáveis em um texto, é importante dizer ao computador quais meta-informações essas variáveis carregam em si. Define-se **meta-informações da variável** (ou do grupo de variável) como informações associadas à variável em questão que não correspondem ao seu valor propriamente. Exemplos de informações são perguntas ao usuário, tipo dos valores dessa variável (texto, data, número, etc), se essa variável exige preenchimento obrigatório ou opcional, se for texto e precisar de uma formatação específica (CPF ou CNPJ, por exemplo), se a resposta deve estar dentro de um conjunto de opções possíveis, entre outras propriedades. O processo de automação, via ferramenta da Looplex, exige que tais informações sejam definidas, para que através de uma entrevista guiada, o usuário consiga montar um documento a partir dessas informações. Chamamos esse processo de 'definição de uma variável e suas propriedades' de **declaração**. Mais uma vez, o modelo ER se apresenta como uma ferramenta poderosa para identificação de entidades e nesse caso, de variáveis e grupos de variáveis.

As variáveis que aparecem no texto são as mais fáceis de serem identificadas. Não obstante, existem ainda aquelas variáveis ocultas ao texto que discriminam fluxos condicionais e outras operações implícitas. É importante que todas as variáveis sejam devidamente discriminadas, catalogadas e especificadas. Se você preferir, você pode especificar dentro dos tópicos as variáveis usadas neles de modo parecido com o diagrama ER. Mas para evitar que o diagrama se sobrecarregue de informações, as especificações podem ser guardadas em uma estrutura chamada de mapa de declarações.

Um **mapa de declarações** é uma estrutura tabular onde todas as meta-informações de uma variável são catalogadas. Retomando o exemplo de elaboração da qualificação via método *bottom-up*, encontramos as seguintes variáveis e grupos de variáveis ali apresentadas. Não se esqueça que o tipo de pessoa também é uma variável, embora não apareça no texto.

Variável
<tipo de pessoa>
<nome>
<nacionalidade>
<estado civil>
<profissão>
<RG>
<CPF>
<razão social>
<CNPJ>
<inscrição estadual>
<:endereço:>

Tabela 3.1: Variáveis de qualificação apresentadas no exemplo do método *bottom-up*

A especificação Looplex define uma lista de propriedades a serem detalhadas para cada variável. São essas as propriedades:

Obrigatoriedade: Essa informação indica se a variável deve ser respondida obrigatoriamente pelo usuário ou não.

Nome: O nome é uma propriedade que identifica a variável amigavelmente. Em um relatório de revisão, isto é, um resumo das respostas dadas pelo usuário, é conveniente usar somente o nome da variável para mapear seu valor.

Pergunta: A pergunta é um texto que será repassado integralmente ao usuário para requisitar o valor da respectiva variável.

Tipo: O tipo de um dado define o conjunto de valores ao qual o dado pertence. O tipo determina o conjunto de valores possíveis que uma variável pode guardar. Toda variável tem um valor que pode ser texto, número, data, horário, lógico (admite apenas sim ou não), listas atômicas (de única escolha), listas não-atômicas (de escolhas múltiplas), imagens ou anexos ou sub-especificações destes: número inteiro, número real, valor monetário, etc. Por exemplo, o valor de um imóvel é um valor monetário, enquanto que a área de um imóvel é medida com números reais.

Opções: Caso o tipo seja uma lista, essa propriedade especifica as opções possíveis para essa variável. Exemplo, tipos de garantia de um contrato de locação: caução; fiança e segura fiança.

Limites: Para os casos numéricos ou listas não-atômicas, pode-se restringir a faixa de valores ou número de respostas possível. O mesmo vale para restringir datas e números de caracteres de um texto. Exemplo: Especificar valores de multa entre 1 e 10% equivale à [0.01, 0.10].

Ajuda: Essa propriedade possibilita que sejam esclarecidas dúvidas a respeito de uma variável específica.

Valor padrão: Você pode também sugerir valores padrão ao usuário do modelo através dessa propriedade. Exemplo: Brasil é um valor padrão de país.

Máscara: Caso seja um campo de texto, você pode também especificar uma máscara para formatação do campo. Exemplo: CPF tem como máscara xxx.xxx.xxx-xx, onde cada x corresponde a um dígito entre 0 e 9.

Para muitos casos, não são aplicáveis algumas propriedades, como por exemplo, máscara a uma variável lógica. Quando for o caso, preencha na tabela N/A. Para casos em que não se deseja definir tal propriedade, deixe-a em branco ou denote por '−'. Retomando a Tabela 3.1 montamos o seguinte mapa de declarações com as especificações observadas na sua respectiva árvore de decisão (ver Figura 3.23):

Variável	É obrigatória?	Nome	Pergunta
<tipo de pessoa>	Sim	Tipo de pessoa	Informe se a parte é PF ou PJ.
<nome>	Sim	Nome	Qual o nome da parte?
<nacionalidade>	Sim	Nacionalidade	Qual a nacionalidade da parte?
<estado civil>	Sim	Estado civil	Qual o estado civil da parte?
<profissão>	Não	Profissão	Qual a profissão da parte?
<RG>	Sim	RG	Qual o RG da parte?
<CPF>	Sim	CPF	Qual o CPF da parte?
<razão social>	Sim	Razão Social	Qual a razão social da parte?
<CNPJ>	Sim	CNPJ	Qual o CNPJ da parte?
<inscrição estadual>	Não	Inscrição estadual	Qual a inscrição estadual da parte?
<:endereço:>	Sim	Endereço	Insira as informações do endereço da parte.

Tabela 3.2: Exemplo de mapa de declaração - Parte I

Nessa primeira tabela de especificação, note que basicamente só mostramos o que foi dado no exemplo anterior. Em casos reais deveríamos considerar muitos outros dados, como dados do cônjuge caso a parte

Variável	Tipo	Opções	Limites
<tipo de pessoa>	Lista atômica	Pessoa física; Pessoa jurídica	N/A
<nome>	Texto	N/A	–
<nacionalidade>	Texto	N/A	–
<estado civil>	Lista atômica	solteiro; casado; separado; divorciado; viúvo	N/A
<profissão>	Texto	N/A	–
<RG>	Texto	N/A	–
<CPF>	Texto	N/A	–
<razão social>	Texto	N/A	–
<CNPJ>	Texto	N/A	–
<inscrição estadual>	Texto	N/A	–
<:endereço:>	Grupo de variáveis	N/A	N/A

Tabela 3.3: Exemplo de mapa de declaração - Partbe II

seja casada (dependendo do regime de bens), dados do representante legal em casos de pessoa incapaz ou relativamente incapaz, RNE para estrangeiros, entre outras informações.

Embora quase todas as especificações não são aplicáveis, vale salientar que o tipo de <:endereço:> é um grupo de variáveis. Naturalmente, é fato que ele também possua sua própria tabela de declaração. Isso implica que deve-se especificar em outro mapa de declaração as variáveis interna de endereço, ou seja, <logradouro>, <número>, <complemento>, <bairro>, <cidade> e <UF>.

Variável	Ajuda	Valor padrão	Máscara
<tipo de pessoa>	–	–	N/A
<nome>	N/A	–	–
<nacionalidade>	–	brasileiro(a)	–
<estado civil>	–	solteiro	–
<profissão>	–	–	–
<RG>	Registro geral	–	–
<CPF>	Cadastro de Pessoas Físicas	–	xxx.xxx.xxx-xx
<razão social>	–	–	–
<CNPJ>	Cadastro Nacional de Pessoa Jurídica	–	xx.xxx.xxx/xxxx-xx
<inscrição estadual>	Consulte o site SINTEGRA	–	xxx.xxx.xxx.xxx
<:endereço:>	–	N/A	N/A

Tabela 3.4: Exemplo de mapa de declaração - Parte III

Lembre-se, quanto mais detalhada for a especificação, melhor o processo de codificação posterior. No entanto, o tempo não para enquanto estiver fazendo a especificação. Calcule bem seu escopo de trabalho. ;-)

Exercício 7. Ajude-nos a especificar o mapa de declaração do endereço. Defina as informações para as variáveis: <logradouro>, <número>, <complemento>, <bairro>, <cidade> e <UF>.

Identificação de operações: Note que determinados casos práticos, algumas variáveis se relacionam entre si de modo a definir relações lógicas entre elas. Por exemplo, suponha que se queira escrever "<nome da pessoa> tem <idade da pessoa> anos de idade." Caso já tenha sido pedido a data de nascimento da pessoa, é redundante pedir também a idade dela. Além disso, corre-se o risco de fornecer informações inconsistentes. Nesse caso, podemos calcular a idade dessa pessoa do seguinte modo: a <idade da pessoa> equivale ao número de anos entre <data de nascimento da pessoa> e <data atual>. Programaticamente seria algo como

<idade da pessoa> = anos(<data de nascimento da pessoa> − <data atual>).

Em um outro exemplo, gostaríamos de tirar do advogado a responsabilidade de fazer contas (é meio perigoso isso, não acha?). Quando se trata de juros compostos nem se fala. Mas graças às operações, esse tipo de preocupação deixa de existir, uma vez que definimos a taxa de juros como

<valor final> = <valor inicial> * (1+<taxa de juros>)^<prazo>,

onde 'a^b' significa 'a' elevado a 'b' para o computador. Assim, as operações oferecem um bom suporte a realização de tarefas críticas no processo de automatização do documento.

Todas essas fórmulas e expressões podem ser gravadas em um mapa lógico, uma espécie de documento anexo onde se pode registrar todas as operações dentro de um modelo sem precisar carregar muito a árvore de decisão.

Exercício 8. Retomando o exemplo da cláusula de contrato social (reescrito a seguir), descreva as operações existentes entre as variáveis identificadas em negrito.

As Partes acordam que, com contagem a partir <**data de contratação**>, data de contratação do Outorgado como prestador de serviços da Companhia ("Período Inicial"), após o decurso do período de 1 (um) ano, o Outorgado ficará *vested em*, ou seja terá o direito de exercer a opção de compra sobre, <**número de quotas**> (<**número de quotas por extenso**>) quotas, sem valor nominal, da Companhia, sendo que a cada período de 1 (um) ano o Outorgado ficará *vested em* um adicional de <**número de quotas adicionais**> (<**número de quotas adicionais por extenso**>) quotas, sem valor nominal, da Companhia, de forma que após decorridos <**período de vesting**> (<**período de vesting por extenso**>) anos do Período Inicial, findo em <**data do fim do vesting**> ("Período de Vesting"), o Outorgado terá o direito de exercer a compra da Participação Total, nos termos do presente Contrato.

Linguagem anotada: Alguns esforços recentes buscam descrever uma linguagem técnica para marcação (rotulação) de textos para que seja possível tornar o processo de automatização de documentos mais simples [27]. Existem aqueles que optam por criar um estilo próprio de marcação; outros que tendem a usar soluções já validadas pelo mercado tecnológico, como a linguagem XML [28], por exemplo. O fato é que a linguagem de marcação tem um objetivo bem definido: descrever uma linguagem padrão compreensível por homens e máquinas, um protocolo transparente em que se anota meta-informações jurídicas e comandos de programação juntamente com o texto final.

No nosso caso, uma vez que temos em mãos todos os recursos suficientes para especificar um modelo de geração automática de documento, tais como *mind map*, *modelo ER*, *árvore de decisão* e *mapas de declaração*, a **linguagem de marcação** emerge como uma linguagem de alto nível (mais próximo à linguagem humana do que de máquina) que não só auxilia na especificação técnica, mas também possibilita a interpretação de modelos mais básicos de automação de documentos pela máquina.

O objetivo da linguagem de marcação é representar os elementos fundamentais da árvore de decisão textualmente, ou seja, títulos de tópicos impressos, estruturas de desvios condicionais e estruturas de repetição. Vamos apresentar três versões de texto anotado: uma primeira versão, simples mas pouco prática, se baseia no uso de sistema de cores para indicar informações invisíveis ao texto e comandos específicos; uma segunda

versão mais resumida, inspirada na linguagem Latex [29], bastante usada em escrita de artigos acadêmicos; e uma terceira versão baseada em XML [28]. Os componentes da árvore de decisão, conforme se mostra na Figura 3.25 podem ser perfeitamente descritos conforme segue:

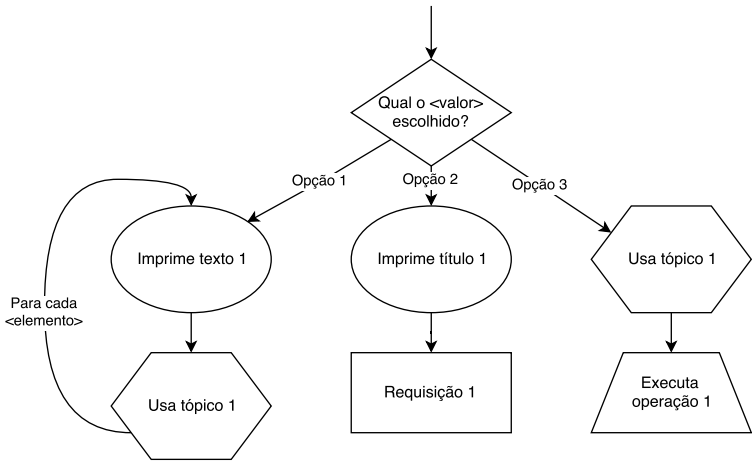

Figura 3.25: Resultado final da árvore de decisão da qualificação com textos.

(I) **Requisição**: Todas as requisições são transparentes em linguagem anotada. Elas não possuem nenhuma representação neste caso.

(II) **Texto**: Um texto é escrito livremente no texto, exatamente como descrito dentro de nós de árvores de decisão.

(III) **Operação**: Para que as operações não sejam impressas, normalmente elas ficam encapsuladas dentro das diretivas de **execute**.

(IV) **Condição**: As componentes de decisão usam diretivas **se** e **senão se** para especificar expressões condicionais (definidas dentro dessas diretivas), e **senão** para condições de escape. Sempre são delimitadas com o termo **fim** para indicar final de bloco de texto.

(V) **Repetições**: Os componentes de repetição usam a diretiva **para cada** para especificar expressões de repetição (definidas dentro dessas diretivas). Também são delimitadas com o termo **fim** para indicar final de bloco de texto.

(VI) **Tópico**: Uma vez que tópicos são declarados externamente, faz-se o uso da diretiva **use** para indicar uma importação de código de outro lugar escrito em linguagem anotada.

Dadas essas definições, a árvore da Figura 3.23 pode ser descrita como linguagem anotada em várias versões:

1. Na primeira versão, usamos cores para destacar cada componente.

```
se <valor> for "Opção 1"
para cada <elemento>
texto 1
use tópico 1
fim
senão se <valor> for "Opção 2"
título 1
senão
```

use tópico 1
execute operação 1
fim

As cores ajudam bastante na identificação dos comandos, mas não é uma boa prática se programar ou especificar com sistemas de cores, visto que as opções de cores não são muitas e a confusão mental que isso traz não justifica seu uso. Por outro lado, especificar com uma linguagem humana não ajuda muito a máquina por falta de formalismo. Por exemplo, a condicional ***se*** *<valor> for "Opção 1"* pode ser escrita de diversas formas em linguagem humana, como por exemplo, *se <valor> for igual a "Opção 1"*, *caso <valor> seja a "Opção 1"*, ou mesmo *<valor> corresponde à "Opção 1"*. Por esta razão se opta por outras formas de representação.

2. Na segunda versão, inspirada em Latex, usa-se comandos iniciados com barras invertidas e expressões próximas de programação, como o símbolo de igualdade (==) e as diretivas traduzidas em inglês[17].

if{<valor> == "Opção 1"}
foreach{<elemento>}
texto 1
use{tópico 1}
end
elseif{<valor> == "Opção 2"}
título 1
else
use{tópico 1}
execute{operação 1}
end

Numa terceira versão se usa XML para anotar o texto. Para que não haja confusão entre variáveis e tags de XML, vamos representar as variáveis como [valor] e [elemento] em vez de <valor> e <elemento>.

<**if conditional**="[valor] == "Opção 1"" >
<**foreach iterator**="[elemento]" >
texto 1
<**use**>tópico 1</**use**>
</**foreach**>
<**elseif conditional**="[valor] == "Opção 2"" >
título 1
<**else**>
<**use**>tópico 1</**use**>
<**execute**>operação 1</**execute**>
</**if**>

Observe a versão Latex da árvore de decisão da Figura 3.23 fazendo o uso de indentação (recuos laterais).

if{<tipo de pessoa> == "Pessoa Física"}

[17] A língua inglesa é usada em programação por ser uma linguagem de código universal e por não possui acentuação.

“<nome>, <nacionalidade>, <estado civil>, ”
if{tem <profissão>} “<profissão>, ” **end**
“portador(a) do documento de identidade (RG) número <rg>, e inscrito no Cadastro de Pessoa Física sob o número <cpf>, domiciliado(a) na <endereço.logradouro>, nº <endereço.número>, ”
if{tem <endereço.complemento>} “<endereço.complemento>, ” **end**
if{tem <endereço.bairro>} “<endereço.bairro>, ” **end**
“<endereço.cidade>, <endereço.uf>, por meio de seu advogado, que esta subscreve ”
else
“<razão social>, inscrita no Cadastro Nacional de Pessoa Jurídica (C.N.P.J.) sob o número <cnpj>, ”
if{tem <inscrição estadual>} “inscrição estadual sob o número <inscrição estadual>, ” **end**
“sediada na <endereço.logradouro>, nº <endereço.número>, ”
if{tem <endereço.complemento>} “<endereço.complemento>, ” **end**
if{tem <endereço.bairro>} “<endereço.bairro>, ” **end**
“<endereço.cidade>, <endereço.uf>, de direito privado, por meio de seu advogado e bastante procurador, que esta subscreve”
end

Observe no exemplo anterior que o texto foi simplesmente copiado para dentro dos blocos (ou corpo) do **if** e do **else**. Um **bloco** é a região entre uma diretiva (**if**, **elseif**, **else** ou **foreach**) e o **end**. O uso do recurso da indentação[18] ajuda a discriminar visualmente quais os comandos. Lembre-se que todas essas diretivas e comandos inseridos dentro do texto não aparecem no texto final produzido pelo automatizador do documento, o *document assembler*. O mesmo vale para as variáveis e grupos de variáveis, sendo que no lugar delas são colocados os valores correspondentes (respostas do usuário, por exemplo). Note que o grupo de variáveis é acessado de modo hierárquico, isto é, acessado nível a nível separado por um ponto. Por exemplo, <endereço.bairro> corresponde a uma variável <bairro> definida dentro do grupo de variáveis <:endereço:>.

Observe ainda que existe trecho do código que se repete dentro do caso de pessoa física e também de pessoa natural. Esse trecho de código é a qualificação do endereço da parte. Uma boa estratégia é aplicar a modularização, como discutida na composição da árvore de decisão. Primeiro se separa o código em outro arquivo correspondente à impressão do endereço e o referencia pelo comando **use** dentro do código, conforme segue o exemplo:

if{<tipo de pessoa> == “Pessoa Física”}
“<nome>, <nacionalidade>, <estado civil>, ”
if{tem <profissão>} “<profissão>, ” **end**
“portador(a) do documento de identidade (RG) número <rg>, e inscrito no Cadastro de Pessoa Física sob o número <cpf>, domiciliado(a) na ”
use{endereço}
“por meio de seu advogado, que esta subscreve ”
else
“<razão social>, inscrita no Cadastro Nacional de Pessoa Jurídica (C.N.P.J.) sob o número <cnpj>, ”
if{tem <inscrição estadual>} “inscrição estadual sob o número <inscrição estadual>, ” **end**
“sediada na ”
use{endereço}
“de direito privado, por meio de seu advogado e bastante procurador, que esta subscreve”

[18] Uso de tabulações no código fonte de um programa para ressaltar ou definir a estrutura do algoritmo.

end

De modo que o texto anotado que define o tópico de endereço é escrito como:

topic{endereço}
“<endereço.logradouro>, nº <endereço.número>, ”
if{tem <endereço.complemento>}
“<endereço.complemento>, ”
end
if{tem <endereço.bairro>}
“<endereço.bairro>, ”
end
“<endereço.cidade>, <endereço.uf>
end

A versão alternativa em XML não é difícil de se derivar, ficando como sugestão para o leitor. Linguagens anotadas baseadas em XML tem uma desvantagem bem grande em relação à versão Latex: é muito verbosa. O termo verboso se refere a quantidade excessiva de código para expressar conceitos mais simples de programação. Note que toda vez que se insere uma tag <xyz> você precisa fechá-la como </xyz>. Isso é uma restrição da linguagem XML. Mas há pessoas que a preferem pela clareza que o código traz. Fica portanto a critério de cada um escolher a linguagem que melhor se identifica.

Exercício 9. Faça um modelo de contrato de compra e venda de automóvel usando um dos padrões de linguagem anotada apresentado. Ao final, especifique cada variável no mapa de declarações.

3.5 Processo de Criação do Modelo de Automação de Documentos: Template

As informações que utilizamos precisam ser fornecidas corretamente pelos usuários. Assim, devemos não só eleger o tipo certo para cada variável ou conjunto, mas devemos fazer as perguntas ou comandos da melhor forma possível. Nossa preocupação com relação ao usuário não é tão formalista, não queremos demonstrar conhecimento jurídico nas perguntas que fazemos a ele ou usar seu conhecimento jurídico. Devemos fazer perguntas simples, diretas e com termos fáceis, de preferência coloquiais e não jurídicos. A lógica jurídica é deixada para as operações e para o texto.

Estruturação de Documentos

Um passo importante da automação de um documento é identificar uma topologia, uma divisão espacial do documento para que possamos separar os trabalhos e modularizar as lógicas. Ao automatizar, por exemplo, uma inicial de ação indenizatória é importante identificar que podemos dividí-la, em (1) endereçamento; (2) descrição geral dos fatos; (3) identificação dos fatos que sustentam a alegação de nexo de causalidade; (4) identificação dos fatos que sustentam a ocorrência de um dano e (5) a caracterização jurídica de um padrão de conduta e indicação dos fatos que correspondem a sua violação; (6) pedidos e (7) assinatura. Dependendo do caso, porém, pode fazer sentido uma divisão mais ampla em (1) endereçamento; (2) fatos; (3) direito; (4)

pedidos e (5) assinaturas. De qualquer forma, devemos buscar as partes com relativo grau de independência entre si para programar por etapas.

Um modelo de automação de documentos, conhecido por **template**, é um programa que implementa todas as regras, operações e cálculos suficientes para concretizar um documento. O *template* é um conjunto de declarações e operações que visa expressar em abstrato todas as possíveis permutações de um documento. Ele é o próprio programa produzido pelo engenheiro jurídico. Ao contrário de um modelo ou exemplo, o *template* comporta diversos caminhos. Um *template* de contrato, por exemplo, expressa esse contrato com qualquer preço, mais de uma forma de pagamento, com ou sem arbitragem, com ou sem garantia, com duas ou mais partes etc. O escopo do *template* é determinado por seu criador. Em geral os escopos são familiares (ex. *template* de inicial trabalhista), mas, por vezes, a similaridade lógica de alguns documentos permite sua reunião como permutações de um mesmo *template* (ex. reunir inicial de ação de cobrança e inicial de execução de título executivo extrajudicial em um mesmo *template*).

Aqui faremos uma breve discussão sobre possíveis etapas de produção de um *template*. Essas etapas são apenas uma sugestão para os iniciantes, bem parecidas com as etapas de definição de árvore de decisão (abordagem *bottom-up*). No entanto, há outras formas possíveis de organizar o processo de automação de um documento.

Coleta de modelos: A forma mais comum de automatizar um documento é começar pela coleta de seus exemplos. É possível conceber um documento abstratamente e programá-lo de forma que suas permutações surjam, pela primeira vez, quando o código está pronto. Porém isso exige um conhecimento muito grande da área. Em geral, "não se sabe que existe uma debênture garantida se só nos deparamos com debêntures de espécie quirografária". Por isso a coleta dos modelos serve para indicar quais são os pontos que mais oscilam de um documento para outro. É a partir dessas permutações que escolheremos o que perguntar para o usuário. Aqui também definimos o escopo do *template* excluindo situações muito excepcionais, já que a engenharia jurídica é, como mencionamos, uma abstração pragmática. Não faz sentido econômico inserir variáveis que são utilizadas apenas em casos extremamente raros. Isso é especialmente verdade tendo em vista que quanto maior o escopo mais difícil é programar o *template*. Assim, selecione apenas o essencial para cobrir a maioria dos casos.

Esboço da lógica: Identificados os modelos e quais são as informações pertinentes, estudamos sua topologia e verificamos, para cada trecho, como cada informação será utilizada em linhas gerais. Para aqueles que ainda se interessam por representações visuais, esse é o momento de desenhar uma árvore ou grafo de decisão. Para aqueles que preferem programar diretamente, faz-se um esboço do código, identificando quais são suas partes e funções.

Declarações: Feito o esboço, seguimos declarando as informações que queremos. Esse é o momento de refletir sobre a qualidade das perguntas, tipos de variáveis etc. É claro que isso pode ser mudado posteriormente, mas mudar declarações depois de já ter feito diversas operações implica ter de desprezar algumas linhas de operações já feitas, uma vez que eram adaptadas à forma declarada anteriormente. É inevitável fazer alterações nas declarações ao longo da programação das operações, mas podemos evitar de refatorar(ou seja, alterar) o código inteiro com planejamento e organização.

Impressões: Declaradas as informações necessárias seguimos fazendo seu uso, que resulta, em última análise na impressão do texto. É aqui que verificamos se cabe uma alegação ou cláusula, que fazemos cálculos dos valores de obrigações, que concatenamos textos, organizamos parágrafos, inserimos citações etc.

Testes: Depois de terminado o *template*, é preciso fazer testes iniciais para avaliar a ocorrência de erros de programação, que serão informados pelo sistema. Em seguida é preciso testar a lógica jurídica, ou seja, ver se os argumentos ou cláusulas impressas são coerentes entre si e com os dados inseridos pelo usuário. Além disso, verificar se o alegado é juridicamente correto. Esse teste mais profundo é idealmente feito por uma terceira pessoa, para evitar o viés de quem programou o *template*.

Manutenção: A manutenção do *template* é uma realidade e faz parte do dia-a-dia do engenheiro jurídico. Invariavelmente haverá um erro no *template*: algum argumento que não foi impresso, um pedido que surgiu sem ser escolhido pelo usuário etc. Logo, esteja atento para o feedback de quem testa ou do usuário final. Como veremos, a organização e limpeza do código facilitam sua própria vida no momento da manutenção. Mesmo que não houvesse erros (e sempre terá!), as teses, os precedentes e a lei mudam. Esteja, portanto, sempre preparado para fazer a manutenção do *template*.

Todos esses procedimentos guiam a produção e manutenção de um *template* desde a parte de especificação técnica até a codificação propriamente. Como a maioria dos trabalhos realizados em equipe, o conflito na edição de documentos pode ser um problema sério. É interessante pensar no uso de softwares para gestão e resolução de conflitos entre documentos editados. Neste sentido, sugerimos o uso de uma ferramenta chamada *git* [30], um sistema de gerenciamento que controla as versões dos códigos de modo distribuído. Os códigos ficam disponíveis em repositórios distribuídos na nuvem, serviços esses oferecidos principalmente pelo Bitbucket[19] e GitHub[20]. Fique atento às vantagens e desvantagens de cada repositório.

[19] Acesse `https://bitbucket.org/`.
[20] Acesse `https://github.com/`.

4 Fundamentos de Lógica e Programação

Para que o leitor se torne um engenheiro jurídico de respeito, é fundamental que se aprenda ferramentas básicas da matemática para uma melhor expressão das abstrações e conceitos jurídicos em linhas de código.

Dentre as ferramentas, serão revisitados conceitos básicos da **Teoria dos conjuntos** (Seção 4.1), bem como introduzidos conceitos elementares de **Lógica formal** (Seção 4.2). Outros conceitos primordiais tais como os **Paradigmas clássicos de programação** (Seção 4.3) e uma breve **Fundamentação algorítmica** (Seção 4.3) também serão discutidos neste capítulo.

4.1 Teoria dos conjuntos

As ideias fundamentais da teoria dos conjuntos e álgebra [31] são provavelmente os conceitos mais importantes da matemática, devido à vasta aplicação em outros campos de conhecimento, e em particular, para as diferentes áreas jurídicas, esses conceitos são fundamentais na estruturação lógica dos documentos. Infelizmente, por se tratar de um conceito básico, pouca atenção é dirigida ao tema. Nesta seção, visitaremos conceitos básicos da teoria de conjuntos, para fornecer ao leitor um conhecimento um pouco mais amplo daquele visto em cursos de matemática de ensino médio.

O que é um conjunto?

Um conjunto é um destes conceitos matemáticos fundamentais que naturalmente entendemos sem qualquer referência a outra ideia matemática. É bem intuitivo assimilar que

Definição 1. *Um conjunto é uma coleção de objetos distintos.*

Os objetos são entidades que podem ser reais ou abstratas. Por exemplo, são objetos do conjunto "sentimentos" o amor, a raiva, o ódio, etc. Além disso, o número desses objetos pode ser finito ou infinito. A única restrição importante é que esses objetos sejam distintos, isto é, objetos unicamente identificáveis.

A ideia de unicidade tem suas nuances. Por exemplo, uma coleção de bolas azuis idênticas dentro de uma urna é um conjunto de bolas, que a princípio são unicamente identificáveis. Neste caso, o conjunto de bolas azuis contém apenas um elemento. Ao passo que, se coloríssemos cada bola com uma cor diferente, ai sim teríamos um conjunto de tantas bolas quanto contiver na urna, pois elas passam a ser não mais unicamente identificáveis.

Num outro exemplo, a sequência de dígitos $3, 4, 1, 2, 7, 2$ não é um conjunto, uma vez que não existe nenhuma forma de distinguir as ocorrências do dígito 2 nessa sequência. Em outras palavras, a ordem dos elementos não importa para a definição de conjunto.

O modo mais básico de definir conjuntos é a partir da *listagem dos elementos delimitados por chaves*: $\{2, 4, 6, 8, 10\}$. Também podemos definir um conjunto por meio de *regras de definição* que identificam a ocorrência de cada objeto, caracterizando-os como membros do conjunto: $\{x : 0 < x < 1\}$. Os dois pontos nesse caso é lido como "tal que". Alguns autores preferem usar a barra ("|") ao invés de dois pontos. Sendo assim, podemos ler o conjunto definido pela regra anterior como "conjunto de valores x tais que x está estritamente entre 0 e 1". Devemos ser capazes de determinar a presença ou a ausência para todos os objetos em relação a um conjunto definido, via de regra. Do contrário, não é válida a regra para definição do conjunto.

Outros modos de definir conjuntos consiste em construí-los a partir de outros conjuntos mais básicos. Discutiremos como fazê-lo depois. O que de fato é relevante nas definições de conjunto são os nomes dados a eles, como forma de referenciá-los em um outro momento. É de praxe denotá-los usando letras maiúsculas, como por exemplo,

$$S = \{x : 0 < x < 1\}.$$

Definição 2. *Um objeto x é um elemento ou membro do conjunto S, denotado por $x \in S$, se x satisfaz a regra de definição de S.*

Podemos escrever que $x \notin S$ se x não é elemento de S.

Definição 3. *O conjunto vazio ou conjunto nulo, denotado por $\emptyset$, é o conjunto que não contém nenhum elemento.*

Uma propriedade importante de um conjunto é o número de elementos que ele contém.

Definição 4. A cardinalidade de S, denotada por $|S|$, é o número de elementos dentro de S.

Exercício 10. Qual a cardinalidade do conjunto $\{0, 1, 2, 3, \ldots, 100\}$?

Note que este número pode ser infinito (assim como em $\{0, 2, 4, 6, \ldots\}$ ou $\{x : 0 < x < 1\}$). Embora exista um formalismo matemático para definições de cardinalidade de conjuntos infinitos, vamos ignorá-las por ora, e assumiremos que a cardinalidade desses conjuntos é simplesmente infinita.

A cardinalidade do conjunto vazio é zero ($|\emptyset| = 0$).

Subconjuntos e Igualdade

Essa seção descreve *relações* entre conjuntos, isto é, conceitos que usamos para comparar dois conjuntos.

Definição 5. *Dois conjuntos S e T são iguais, denotado por $S = T$, se S e T contém exatamente os mesmos elementos.*

Esse conceito é muito trivial para aborrecer o leitor, mas em matemática, é importante haver uma concordância até mesmo com os conceitos mais simples. Todas as definições subsequentes são embasadas nessa definição, de modo que só é capaz de compreender ideias mais complicadas com clareza se entendermos as ideias simples que as fundamentam.

Exercício 11. Os seguintes conjuntos $\{0, 1, 5, 7\}$ e $\{0, 7, 5, 1\}$ são iguais?

Definição 6. *Um conjunto S é um subconjunto de um outro conjunto T, denotado por $S \subset T$ se todo elemento de S também é elemento de T.*

Observe que, por esta definição, $S \subset T$ não se exclui a possibilidade de que todo elemento de T seja também elemento de S.

Definição 7. *Um conjunto S é um subconjunto próprio de T se $S \subset T$ e $S \neq T$.*

Neste caso, todo elemento de S está também em T, mas existe algum elemento de T que não está em S.

Observação quanto à notação: Alguns autores escrevem $S \subseteq T$ para denotar subconjuntos que podem não ser próprios e $S \subset T$ para denotar subconjuntos próprios. Outros usam $S \subset T$ para denotar subconjuntos que podem não ser próprios e $S \subsetneq T$ para denotar subconjuntos próprios. Além dessas, existem outras combinações de notação também.

Tenha certeza que você reconhece qual convenção adotada pelo autor em seus estudos, além de escolher uma convenção consistente em suas próprias notações. Para os fins deste estudo, adotaremos a notação de $S \subset T$ para denotar subconjuntos que podem não ser próprios.

Se R, S e T são dois conjuntos quaisquer, valem as seguintes propriedades:

- $\emptyset \subset S$
- $S = S$ (reflexividade)
- $S \subset S$ (reflexividade)
- *Se* $S = T$, *então* $T = S$ (simetria)
- *Se* $R = S$ *e* $S = T$ *então* $R = T$ (transitividade)
- *Se* $R \subset S$ *e* $S \subset T$, *então* $R \subset T$ (transitividade)
- $S \subset T$ *e* $T \subset S$ *se, e somente se,* $S = T$
- *Se* $S \subset T$, *então* $|S| \leq |T|$, *e se* $S \subsetneq T$, *então* $|S| < |T|$

Interseções e Uniões

Essa seção e a consequente discutiremos operações sobre conjuntos, a saber, maneiras as quais os conjuntos podem ser combinados a fim de formar novos conjuntos.

Definição 8. A interseção de dois conjuntos S e T, escrita como $S \cap T$ é o conjunto de elementos comuns a ambos S e T, isto é,

$$S \cap T = \{x : x \in S \text{ e } x \in T\}.$$

Esta definição se lê como: o conjunto de objetos x tais que x é um elemento de S e x é um elemento de T.

Exemplo 1. Se $S = \{e_1, e_3, e_4, e_5, e_6\}$ e $T = \{e_0, e_2, e_4, e_6, e_12\}$ então $S \cap T = \{e_4, e_6\}$.

Exercício 12. Se $S = \{e_4, e_7, e_8, e_15\}$ e $T = \{e_4, e_7, e_12, e_13\}$ então qual é a cardinalidade de $S \cap T$?

Exemplo 2. Se $S = \{x : 0 \leq x \leq 1\}$ e $T = \{x : 0.5 < x < 2\}$, então $S \cap T = \{x : 0.5 < x \leq 1\}$.

Definição 9. A união de dois conjuntos S e T, escrita como $S \cup T$, é o conjunto de elementos que estão ou S ou T ou em ambos, isto é,

$$S \cup T = \{x : x \in S \text{ ou } x \in T\}.$$

Note que quando duas sentenças são conectadas com "ou", isto significa que pelo menos uma das sentenças seja verdade.

Exemplo 3. Se $S = \{e_1, e_2, e_3, e_4, e_5, e_6\}$ e $T = \{e_0, e_2, e_4, e_8\}$ então $S \cup T = \{e_0, e_1, e_2, e_3, e_4, e_5, e_6, e_8\}$.

Exercício 13. Se $S = \{e_2, e_6, e_7, e_9\}$ e $T = \{e_0, e_1, e_3, e_5\}$ então qual é o conjunto de $S \cup T$?

Exemplo 4. Se $S = \{x : 0 \leq x \leq 1\}$ e $T = \{x : 0.5 < x < 2\}$, então $S \cup T = \{x : 0 \leq x < 2\}$.

Pode-se combinar múltiplas operações para criar novos conjuntos a partir de outros. Em alguns casos, dois diferentes modos de combinação de conjuntos produzem o mesmo resultado. Neste caso, é possível substituir uma combinação por outra dentro da equação.

Se S é um conjunto construído a partir de outros, podemos substituir S dentro da expressão pela expressão que descreve a construção de S (Ao fazer essas substituições, é recomendável colocar parênteses ao redor da respectiva expressão). Ao repetir aplicações deste princípio é possível ter "*insights*" reveladores a respeito do relacionamento de certos conjuntos.

As regras de substituição com respeito às uniões e interseções são exibidas abaixo. Com uma pequena reflexão você logo se convencerá da validade da maioria delas.

- $S \cap S = S \cup S = S$ (Idempotência)

- $S \cup \emptyset = S$, assim como $S \cap \emptyset = \emptyset$ (Elemento Neutro)
- $S \cup T = T \cup S$, assim como $S \cap T = T \cap S$ (Propriedade Comutativa)
- $R \cup (S \cup T) = (R \cup S) \cup T$, assim como $R \cap (S \cap T) = (R \cap S) \cap T$ (Propriedade Associativa)
- $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$ (Distributiva $-$ $\cup$ sobre $\cap$)
- $R \cap (S \cup T) = (R \cap S) \cup (R \cap T)$ (Distributiva $-$ $\cap$ sobre $\cup$)
- $S \cup (S \cap T) = S \cap (S \cup T) = S$
- $S \subset S \cup T$, assim como $S \cap T \subset S$
- $S \subset T$ *se, e somente se,* $S \cup T = T$.
- $If R \subset T$ *e* $S \subset T$, *então* $R \cup S \subset T$.
- $If R \subset S$ *e* $R \subset T$, *então* $R \subset S \cap T$.
- $If S \subset T$ *então* $R \cup S \subset R \cup T$ *e* $R \cap S \subset R \cap T$.
- $S \cup T \neq \emptyset$ *se, e somente se,* $S \neq \emptyset$ *ou* $T \neq \emptyset$.
- $If S \cap T \neq \emptyset$, *então* $S \neq \emptyset$ *e* $T \neq \emptyset$.
- $S = T$ *se, e somente se,* $S \cup T \subset S \cap T$.

A propriedade chave com relação a cardinalidade de uniões e intercessões é

$$|S \cup T| = |S| + |T| - |S \cap T|.$$

Isto é, o número de elementos da união entre dois conjuntos é a soma do número total de elementos de cada um deles, menos o número de elementos que aparecem duplicados nestes dois conjuntos.

Exemplo 5. Se $S = \{e_1, e_2, e_3, e_4, e_5, e_6\}$ e $T = \{e_0, e_2, e_4, e_6, e_8\}$ então $S \cup T = \{e_0, e_1, e_2, e_3, e_4, e_5, e_6, e_8\}$ e $S \cap T = \{e_2, e_4, e_6\}$. Então $|S| = 6$ e $|T| = 5$. Como $|S \cap T| = 3$, então existem três elementos duplicados. Quando unimos S e T, combinamos as duas listas de elementos, mas cada elemento duplicado é contado apenas uma vez. Assim, $|S \cup T| = 6 + 5 - 3 = 8$.

Exercício 14. Se $S = \{e_3, e_8, e_9, e_10\}$ e $T = \{e_0, e_1, e_3, e_8\}$ então qual é a cardinalidade do conjunto de $S \cup T$?

Se dois conjuntos não têm elementos em comum, esta fórmula se torna ainda mais simples.

Definição 10. *Dois conjuntos* S *e* T *são mutuamente exclusivos se* $S \cap T = \emptyset$.

Se S e T são mutuamente exclusivos, então $|S \cap T| = 0$, logo $|S \cup T| = |S| + |T|$.

Exemplo 6. Se $S = \{e_1, e_3, e_5\}$ e $T = \{e_0, e_2, e_4, e_6, e_8\}$ então $S \cup T = \{e_0, e_1, e_2, e_3, e_4, e_5, e_6, e_8\}$ e $S \cap T = \emptyset$. Logo $|S| = 3$ e $|T| = 5$. Mas $|S \cap T| = 0$, desde que nenhum elemento aparece em ambas as listas. Ao compormos a união, combinamos as duas listas de elementos, e por não haverem duplicatas, temos que $|S \cup T| = 3 + 5 = 8$.

Diferença entre conjuntos e complementos

Outro modo de construir novos conjuntos a partir de outros, é pela remoção de alguns elementos do conjunto e considerarmos os elementos que estão fora deste conjunto.

Definição 11. *A diferença entre S e T, escrita como $S \setminus T$*, é o conjunto de elementos que estão em S mas não em T:

$$S \setminus T = \{x : x \in S \text{ e } x \notin T\}.$$

Exemplo 7. Se $S = \{e_1, e_2, e_3, e_4, e_5, e_6\}$ e $T = \{e_0, e_2, e_4, e_6, e_8\}$ então

$$S \setminus T = \{e_1, e_3, e_5\}, \text{ mas } T \setminus S = \{e_0, e_8\}.$$

Exemplo 8. Se $S = \{e_1, e_3, e_5\}$ e $T = \{e_0, e_2, e_4, e_6, e_8\}$ então

$$S \setminus T = S \text{ assim como } T \setminus S = T.$$

Exemplo 9. Se $S = \{x : 0 \leq x \leq 1\}$ e $T = \{x : 0.5 < x < 2\}$, então

$$S \setminus T = \{x : 0 \leq x \leq 0.5\} \text{ e } T \setminus S = \{x : 1 < x \leq 2\}.$$

Exercício 15. Se $S = \{e_1, e_3, e_4, e_6\}$ e $T = \{e_0, e_2, e_4, e_6, e_8\}$ então determine $S \setminus T$ e $T \setminus S$.

Como podemos observar, em geral, $S \setminus T \neq T \setminus S$ (a diferença de conjuntos não é comutativa).

Duas importantes propriedades de diferenças de conjunto são:

1. $S \setminus T = S \setminus (S \cap T)$
2. $|S \setminus T| = |S| - |S \cap T|$

A primeira propriedade afirma que os elementos de T que não estão em S não participam da regra de definição de $S \setminus T$. A segunda propriedade afirma que o número de elementos removidos de S para produzir $S \setminus T$ é o mesmo número de elementos que S e T têm em comum.

Definição 12. *A diferença simétrica entre S e T é $(S \cup T) \setminus (S \cap T)$, denotado por $S \triangle T$ isto é, o conjunto de elementos em S ou T, mas não em ambos.*

A cardinalidade de uma diferença simétrica satisfaz a seguinte propriedade:

$$|S \triangle T| = |S| + |T| - 2|S \cap T|$$

Exercício 16. Se $S = \{e_1, e_3, e_4, e_6\}$ e $T = \{e_0, e_2, e_4, e_6, e_8\}$ então determine $S \triangle T$ e sua respectiva cardinalidade.

A contagem das dos conjuntos S e T exclui todas as cópias duplicadas de ambos os conjuntos.

Exemplo 10. Se $S = \{e_1, e_2, e_3, e_4, e_5, e_6\}$ e $T = \{e_0, e_2, e_4, e_6, e_8\}$ então

$$S \triangle T = \{e_0, e_1, e_3, e_5, e_8\}, \text{ e } |S \triangle T| = 6 + 5 - 2 * 3 = 5.$$

Exemplo 11. Se $S = \{x : 0 \leq x \leq 1\}$ e $T = \{x : 0.5 < x < 2\}$, então

$$S \triangle T = \{x : 0 \leq x \leq 0.5 \text{ ou } 1 < x \leq 2\}.$$

Em muitos problemas de conjuntos, todos os conjuntos são definidos como subconjuntos de algum conjunto de referência. Este conjunto referência é chamado *universo*. Mantenha sempre em mente nessa seção que iremos nos referir ao conjunto universo U, e assumiremos que todos os outros conjuntos R, S, e T são subconjuntos de U.

Definição 13. *Em relação ao universo U, o complemento de S, denotado por S'* é o conjunto de todos os elementos do universo não contidos em S, isto é,

$$S' = \{x : x \subset U \text{ e } x \notin S\}$$

Outra notação comum para o complemento de S é $\bar{S}$. Complementos de expressões de conjuntos podem ser escritas de forma similar, por exemplo: $(S \cap T)'$ ou $S \bar{\cap} T$.

Uma propriedade importante com relação a cardinalidade é

$$|S'| = |U| - |S|$$

Exemplo 12. Seja U o conjunto de números naturais de 1 a 10, e $S = \{2, 4, 6, 8\}$. Determine o conjunto S' e sua respectiva cardinalidade.

Exemplo 13. Se $U = \{e_0, e_1, \ldots, e_{10}\}$ e $S = \{e_1, e_2, e_3, e_4, e_5, e_6\}$, então $S' = \{e_0, e_7, e_8, e_9, e_{10}\}$. Também, $|U| = 11$ e $|S| = 6$, logo $|S'| = 11 - 6 = 5$.

Exemplo 14. Se $U = \{x : x \text{ é um número real}\}$ e $S = \{x : 0 \leq x \leq 1\}$, então $S' = \{x : x < 0 \text{ ou } x > 1\}$.

A seguir, mostraremos importantes propriedades de complementos e diferenças de conjuntos.

- $U' = \emptyset \ e \ \emptyset' = U$
- $S \cup S' = U \ e \ S \cap S' = \emptyset$
- $S \cup U = U \ e \ S \cap U = S$

- $S'' = S$
- $S = T'$ *se, e somente se,* $T = S'$.
- $S \subset S'$ *se, e somente se,* $S = \emptyset$.
- $S \subset T$ *se, e somente se,* $T' \subset S'$.
- $S \setminus T = S \cap T'$
- $(S \cup T) \setminus (S \cap T) = (S \setminus T) \cup (T \setminus S)$
- $(S \cup T)' = S' \cap T'$
- $(S \cap T)' = S' \cup T'$

As últimas duas propriedades são conhecidas como leis de Morgan. Elas fornecem um modo útil de converter expressões arbitrárias em outras que envolvem apenas complementos e uniões ou somente complementos e interseções.

Quanto à ordens de execução, todas as operações são feitas antes das comparações, sendo que as operações entre parênteses são executadas antes das outras. As operações de complemento são realizadas antes de qualquer outra operação de conjunto. Os parênteses são obrigatórios em torno das expressões as serem complementadas.

4.2 Lógica formal

Com o advento da Internet, o ser-humano desenvolveu uma habilidade bem peculiar: a atenção multifocal. Devido a plasticidade cerebral do ser-humano da atualidade, ele consegue manter a atenção em diversos pontos de concentração, e com isso, absorve uma quantidade considerável de conhecimento. No entanto, a profundidade dessa assimilação é bastante questionável. Hoje em dia é difícil encontrar pessoas que não recorram à calculadora para efetuar operações aritméticas elementares. A tabuada tem se tornado um dispositivo de aprendizado tão obsoleto como o ábaco. Por estas e outras, vamos "desenferrujar" alguns neurônios de modo a preparar o operador do Direito a modelar problemas complexos que envolvem o Direito.

Se a lógica é de domínio das exatas, surge a questão: porque não ensinar o Direito para um programador, ao invés de ensinar lógica de programação aos operadores do Direito? A resposta está na inércia, o custo de aprendizado, os pontos de interesse de cada área, entre outros argumentos. É mais conveniente estruturar um conhecimento fundamentado em lógicas "informalmente organizadas" nas cabeças dos profissionais do Direito do que ensinar conceitos jurídicos a um programador. A lógica em si mesma é uma ferramenta, não um tema de concentração. Assim, buscamos apresentar ao leitor uma visão mais ampla da lógica formal, conforme se apresenta a seguir.

A lógica [32] é bastante difundida em disciplinas exatas (matemática e ciência da computação) e de filosofia. Se refere ao estudo racional de argumentos, avaliando a relação entre todos os termos integrantes de uma sentença. É conveniente estruturar o raciocínio a partir de teorias de lógica formal, conhecida como lógica simbólica.

Diversas vertentes de lógica formal compartilham uma mesma necessidade de estruturação do raciocínio

em torno de definições e axiomas fundamentais. Um bom exemplo disso é o uso da lógica modal [33], que procura lidar com modalidades (tratar de modos quanto a tempo, possibilidade, probabilidade, etc) para estruturação lógica. Tradicionalmente, as sentenças são expressas em termos de necessidade N_a (lido como "necessário que se ocorra a") e possibilidade P_a (lido como "possível que se ocorra a").

Em alguns ramos do ireito, como no caso de automação de agentes judiciários virtuais, outros tipos de lógica modal são empregadas. A lógica deôntica se mostra uma ferramenta promissora na interpretação de sentenças e resoluções de conflitos. Em lógica deôntica a relação entre dever e poder é estabelecida:

O_a: obrigatório que se ocorra a (relação de dever)

P_a: permitido que se ocorra a (relação de poder)

F_a: proibido que se ocorra a

L_a: facultativo que se ocorra a

Na prática, a corresponde a uma sentença arbitrária que descreve uma ação ou o conjunto delas. Para se ter noção da utilidade da lógica formal, podemos intuitivamente deduzir que: L_a é equivalente a P_a ou $P_{\neg a}$, onde $\neg a$ significa a não ocorrência de a.

Outro tipo de lógica modal bastante usada no meio jurídico é a lógica epistêmica, a qual aborda aspectos argumentativos em relação a certeza e incerteza de eventos descritos por meio de sentenças. Neste caso,

S_a: é certo que se ocorra a (relação de certeza)

R_a: é possível que se ocorra a (relação de incerteza)

Na prática, podemos relacionar um operador com outro de modo que S_a equivale a $\neg P_{\neg a}$. Alguns exemplos que expressam possibilidade: "pode ser que haja vida em outros planetas, mas não se pode provar", "não se pode saber se duendes existem ou não". Outros expressam certeza: "ao nível do mar, é impossível a existência de gelo a 100^o Celsius", ou "é certo que 100Kg de algodão pesam o mesmo que 100Kg de chumbo".

Dentre as lógicas formais existentes, adotaremos a lógica proposicional para o desenvolvimento do raciocínio lógico dos engenheiros jurídicos, por se tratar de um formalismo bem estabelecido em disciplinas de computação [34].

O que é lógica proposicional?

A lógica proposicional [35] é um formalismo usado na representação de **proposições** que podem ser compostas por meio de combinações de outras proposições **atômicas** concatenadas por **conectores lógicos** e um sistema de **regras de derivação**, os quais nos permite elaborar fórmulas com um sentido mais amplo (os teoremas) do sistema formal.

De um modo sintético, a lógica proposicional é um sistema formal composto por (a) uma *linguagem formal*, usada para representar conhecimento; e (b) por *métodos de inferência*, usados para representar raciocínio. Tem como finalidades principais (i) a representação de argumentos, isto é, sequências de sentenças em que uma delas é uma conclusão e as demais são premissas; bem como (ii) a validação de argumentos, ou seja, a verificação se uma conclusão é consequência lógica de suas premissas.

Exercício 17. Intuitivamente, qual dos dois argumentos a seguir é válido?

a) Se neva, então faz frio. Está nevando. Logo, está fazendo frio.

b) Se chove, então a rua fica molhada. A rua está molhada. Logo, choveu.

Proposição

Cada argumento é expresso por uma proposição. Uma proposição é uma sentença declarativa que pode ser verdadeira ou falsa, mas não as duas coisas ao mesmo tempo.

Exercício 18. Quais das sentenças a seguir são proposições?

a) Por gentileza, abra a porta.

b) Esta semana tem oito dias.

c) Em que continente fica o Brasil?

d) A Lua é um satélite da Terra

De um modo geral, uma proposição é uma sentença verificável com "sim" e "não".

Exercício 19. Por que a sentença "esta frase é falsa" não é uma proposição?

Conectivos

As proposições podem ser combinadas de modo a formar novas proposições mais completas em seu sentido. Assim como na língua natural, podemos concatenar sentenças por meio de conectores (e, ou, mas, porém, etc). No caso da lógica, tais conectores são chamados de conectivos lógicos, ou simplesmente, conectivos.

Conectivos são partículas (**não**, **e**, **ou**, **então**, **se e somente se**) que permitem construir sentenças complexas a partir de outras mais simples. Tais partículas nos permitem concatenar preposições atribuindo novos sentidos lógicos ao resultado final concatenado.

Como citado na definição de lógica proposicional no início desse capítulo, é possível distinguir uma proposição atômica de uma composta pelo emprego dos conectivos. Sendo assim, uma proposição atômica não tem nenhum conectivo, enquanto que as compostas os empregam. Por exemplo, a partir das premissas (proposições atômicas) "Está chovendo" e "A rua está molhada" podemos construir as sentenças (proposições compostas) "Se está chovendo, então a rua está molhada.", ou ainda, "Não está chovendo".

Exercício 20. Considere a seguinte sentença "Se está chovendo, então a rua está molhada". Se eu te dissesse que "Não está chovendo", qual seria sua conclusão?

Notação formal

O formalismo é um conjunto de normas e regras rígidas a serem seguidas a fim de facilitar a comunicação inequívoca entre as partes comunicantes. A fim de expressarmos sentenças de modo compreensível e compacto, é necessário a assimilação de uma notação formal para análise e validação da lógica proposicional. Para tanto, usamos uma sintaxe específica para representação das sentenças lógicas. **Sintaxe** é um conjunto de regras que delimita a linguagem sobre a qual se estruturam as sentenças. Ela é composta por um conjunto de símbolos para representar não só as proposições, mas os conectivos e fórmulas proposicionais.

Os conectivos podem ser representados pelos símbolos $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$, ordenados por precedência. Quanto às fórmulas, considere inicialmente que todas as proposições são fórmulas. Em geral, as proposições são representadas por letras minúsculas e fórmulas por letras do alfabeto grego. Assim, Se α e β são fórmulas, então também são fórmulas:

A negação de α, ou seja, **não** α: $\neg\alpha$,

A conjunção de α e β, ou seja, α **e** β: $\alpha \wedge \beta$,

A disjunção de α e β, ou seja, α **ou** β: $\alpha \vee \beta$,

A implicação entre α e β, ou seja, **se** α **então** β: $\alpha \rightarrow \beta$.

A equivalência entre α e β, ou seja, α **se, e somente se** β: $\alpha \leftrightarrow \beta$.

A seguir apresentamos algumas leituras que a negação, conjunção, disjunção, implicação e bi-implicação (ou equivalência) podem ter na linguagem natural [35].

Negação: $\neg\alpha$: A negação pode ser lida em linguagem natural como:

- Não α
- Não se dá que α
- Não é fato que α
- Não é verdade que α
- Não é que α
- Não se tem α

Conjunção: $\alpha \wedge \beta$: A conjunção pode ser lida em linguagem natural como:

- α e β;
- α, embora β;
- α, assim como β;
- α e, além disso, β;

- Tanto α como β;
- α e também β;
- Não só α, mas também β;
- α, apesar de β.

Disjunção: $\alpha \vee \beta$**:** A disjunção pode ser lida em linguagem natural como:

- α ou β ou ambos;
- α ou β.

Implicação: $\alpha \rightarrow \beta$**:** A implicação pode ser lida em linguagem natural como:

- Se α, então β;
- Se α, isto significa que β;
- Tendo-se α, então β;
- Quando α, então β;
- α só se β;
- α somente quando β;
- α, só no caso de β;
- α implica β;
- α acarreta β;
- α é condição suficiente para β;
- β é condição necessária para α
- β, sempre que se tenha α;
- β, contanto que α.

Bi-implicação: $\alpha \leftrightarrow \beta$**:** A bi-implicação pode ser lida em linguagem natural como:

- α se e só se β;
- α se e somente se β;
- α quando e somente quando β;
- α equivale a β;
- α é condição necessária e suficiente para β.

Exercício 21. Escreva as sentenças a seguir em linguagem simbólica, usando sentenças básicas (ou atômicas), isto é, as sentenças que não podem ser construídas a partir de outras sentenças.

a) Se Antônio está feliz, a esposa do Antônio não está feliz, e se o Antônio não está feliz, a esposa do Antônio não está feliz.

b) Ou Antônio virá à festa e Pedro não, ou Antônio não virá à festa e Pedro se divertirá.

c) Uma condição necessária e suficiente para o rei ser feliz é ele ter vinho, mulheres e música.

d) Teresa vai ao cinema só se o filme for uma comédia.

Para que haja avaliação de uma fórmula, é necessário compreender a semântica das sentenças que a compõe. Semântica é o valor ou significado das sentenças em termos lógicos, podendo ser *verdadeiro* (V) ou *falso* (F).

A interpretação de uma ou mais proposições lógicas se dá pela análise combinatória de todas as possibilidades. deste modo, uma fórmula contendo n proposições admite 2^n interpretações distintas. Quando para todas as interpretações possíveis de uma fórmula se constata resultado verdadeiro, dizemos que a fórmula é **válida** (ou **tautológica**), ou seja, é verdadeira em toda interpretação. Quando para alguma interpretação de uma fórmula se constata resultado verdadeiro, dizemos que tal fórmula é **satisfatível** (ou **contingente**), isto é, é verdadeira em alguma interpretação. Do contrário, se para todas as interpretações possíveis de uma fórmula se constata resultado falso, dizemos que esta fórmula é **insatisfatível** (ou **contraditória**), isto é, é verdadeira em nenhuma interpretação.

Tabela-verdade

Um mecanismo poderoso na avaliação proposicional é a tabela-verdade. A tabela-verdade é um sistema tabular onde a primeira linha contém uma fórmula, e cada uma das demais linhas da tabela representa uma interpretação lógica, denominada *valoração*.

É comum que as fórmulas sejam representadas desde suas premissas atômicas até a fórmula objetivo de modo construtivo. Observe um exemplo de tabela-verdade usada na avaliação de duas proposições p e q concatenados por conjunção (**e** lógico).

p	q	$p \wedge q$
V	V	V
V	F	F
F	V	F
F	F	F

Tabela 4.5: Conjunção entre p e q

De um modo geral, com a tabela da verdade é possível determinar o valor de uma fórmula a partir de simples valorações aplicadas a cada proposição presente nesta.

Em uma mesma tabela é possível avaliar várias fórmulas em paralelo. Observe a seguinte tabela-verdade usada na avaliação de duas proposições p e q compostas pelos conectivos até aqui apresentados, a saber, conjunção, disjunção, negação e implicação:

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$
V	V	F	V	V	V	V
V	F	F	F	V	V	F
F	V	V	F	V	F	F
F	F	V	F	F	V	V

Tabela 4.6: Conectivos aplicáveis em p e q

Apenas algumas observações para desmantelar sofismas quanto a operações lógicas, p ou q é válido quando um dos elementos é verdadeiro. Se ambos forem verdadeiros, a operação p ou q continua válida, uma vez que um dos elementos é verdadeiro. O vício de linguagem causado pelo uso da partícula "ou" como operando de exclusividade mútua não é considerado válido. Analogamente, a implicação é comumente confundida como bi-implicação. Redobre a atenção quando esses conectivos ocorrerem.

Exercício 22. Preencha a tabela-verdade abaixo.

p	q	r	$\neg p$	$\neg p \wedge q$	$(\neg p \wedge q) \vee r$	$(\neg p \wedge q) \vee r \rightarrow q$
V	V	V				
V	V	F				
V	F	V				
V	F	F				
F	V	V				
F	V	F				
F	F	V				
F	F	F				

Representação do conhecimento

A representação do conhecimento é um processo de formalização lógica muito útil para expressar o raciocínio com precisão. O processo para compor sentenças formais compreende:

1. Identificação das palavras da sentença que correspondem a conectivos.
2. Identificação das partes da sentença que correspondem as proposições atômicas e associamos a cada uma delas um símbolo proposicional.
3. Elaboração da fórmula correspondente à sentença, substituindo suas proposições atômicas pelos respectivos símbolos proposicionais e seus conectivos lógicos pelos respectivos símbolos conectivos.

O objetivo deste material com a representação do conhecimento é desenvolver a capacidade do engenheiro formalizar, formatar, estruturar seus conhecimentos jurídicos em uma linguagem matematicamente compreensível não só por seres humanos, mas por máquinas também. Considere as seguintes sentenças:

1. "Está chovendo."
2. "Se está chovendo, então a rua está molhada."
3. "Se a rua está molhada, então a rua está escorregadia."

Conforme propomos, primeiramente destacamos os conectivos em "**Se** está chovendo**, então** a rua está molhada." e em "**Se** a rua está molhada**, então** a rua está escorregadia.". Sendo assim, podemos destacar as proposições atômicas, associando a eles um símbolo proposicional respectivo:

p : "Está chovendo."

q : "A rua está molhada."

r : "A rua está escorregadia."

Finalmente, concebemos uma formalização que constitui nossa *base de conhecimento*:

$$\Delta = \{p, p \to q, q \to r\}$$

Cada argumento que compõe nossa base de conhecimento é implicitamente concatenado mediante a conjunção (representado pela vírgula).

Embora o leitor possa se assustar um pouco com a notação, na prática, ele está habituado a fazê-lo implicitamente. A formalização e constituição de bases de conhecimento dentro de um documento jurídico é uma tarefa bastante comum, visto que definições de termos presentes em documentos jurídicos num glossário, bem como sua referência ao longo do texto é um típico procedimento de composição de vocabulário. Isto é, quando você se depara com um p ou um q, você deve associá-los ao conhecimento sumarizado a eles. É como se p e q fossem termos definidos que simbolizam todo um conceito associado.

Exercício 23. Simbolize as sentenças abaixo, dado o seguinte esquema:

p : "O estudante comete erros."

q : "Há motivação para o estudo."

r : "O estudante aprende a matéria."

a) Se o estudante não comete erros, então ele aprende a matéria.

b) Se não há motivação para o estudo, então o estudante não aprende a matéria.

c) Se há motivação para o estudo, o estudante não comete erros.

d) O estudante aprende a matéria se, e somente se, há motivação para o estudo.

Muitas vezes é preciso desdobrar uma sentença descrita em português para que seja possível a representação formal do conhecimento. Por exemplo, "Se ele for à festa hoje eu irei, senão não irei." pode ser desdobrado em "Se ele for à festa hoje então eu irei à festa hoje", assim como "Se ele não for à festa hoje então eu não irei à festa hoje". Definindo o esquema: p : "Ele vai à festa hoje" e q : "Eu vou à festa hoje", a base do conhecimento resultante da análise das sentenças é expresso como:

$$\Delta = \{p \to q, \neg p \to \neg q\}.$$

Exercício 24. Defina o esquema e simbolize as sentenças abaixo:

a) Ou Capitu é ou não é a criação mais notável de Machado de Assis.

b) Não é verdade que Machado de Assis escreveu ou não escreveu poesias.

c) Se é fácil ler o que José da Silva escreveu, não é fácil ler o que escreveu Guimarães Rosa.

A base de conhecimento por si só corresponde um conjunto de regras a serem avaliadas. Um *argumento* por sua vez, é uma sequência de premissas seguida de uma conclusão. Nesse caso, uma argumentação é a concatenação de premissas aplicada sobre uma base de conhecimento específica.

Por exemplo, "Se neva, então faz frio" e "Está nevando". "Logo, está fazendo frio". Em suma, se construíssemos um vocabulário que define n : "neve" e f : "frio", a avaliação da base do conhecimento $\{n \to f, n\}$ resulta em f. Formalmente, esse argumento é expresso com o símbolo de conclusão $\vDash$, conforme segue:

$$\{n \to f, n\} \vDash f.$$

Costuma-se separar as premissas da base do conhecimento para que haja uma distinção clara do que são hipóteses, o que são validades dedutivas (inferências), e o que são conclusões. Assim, o argumento também pode ser expresso como segue:

$$n : \{n \to f\} \vDash f.$$

Exercício 25. Usando a sintaxe da lógica proposicional, formalize o argumento:

- Se o time joga bem, então ganha o campeonato.
- Se o time não joga bem, então o técnico é culpado.
- Se o time ganha o campeonato, então os torcedores ficam contentes.
- Os torcedores não estão contentes.
- Logo, o técnico é culpado?

A partir da consolidação de argumentos clássicos é possível ter em mãos um conjunto de regras a serem aplicadas em casos reais. Por exemplo, o exercício apresentado faz o uso de uma propriedade conhecida como **contrapositiva**, que diz que $p \to q$ é equivalente à $\neg q \to \neg p$.

Formas de argumentos básicos e derivados

Embora seja bem conhecida, a *dislexia simbólica*, que assombra milhares de profissionais do Direito, o presente material acredita que raciocinar de modo simbólico ajuda ao projetista jurídico a ter uma visão mais abstrata a respeito do próprio pensamento jurídico em si. Sendo assim, trataremos os argumentos lógicos a princípio de forma simbólica, e na sequência, usaremos um exemplo cotidiano de aplicação prática

de cada argumento.

Dentre os argumentos consolidados, selecionamos os mais úteis para aplicações afins. Em todos os casos, considere duas proposições arbitrárias p e q.

Modus: Esse tipo de argumentação considera a checagem de expressões condicionais. O *Modus Ponens*

$$(p \to q) \wedge p \vDash q$$

é a base da avaliação condicional que nos permite tomar decisões, pois averígua se a condição p implica q e se averígua p, consequentemente ocorre q. Exemplo: "Se (p) chover (q) eu vou pra casa. Choveu! Logo, fui pra casa".

O *Modus Tollens*

$$(p \to q) \wedge \neg q \vDash \neg p$$

é a base da contrapositiva, que nos permite confrontar uma hipótese mediante a contradição (prova por absurdo), pois averígua se a condição p implica q e não se averígua q, consequentemente não ocorre q. Exemplo: "Se (p) chover (q) eu vou pra casa. Não fui pra casa! Logo, não choveu".

Silogismo: O curso de lógica jurídica tem uma forte fundamentação no silogismo. O silogismo é um tipo de argumentação que objetiva a simplificação de hipóteses. O *Silogismo Hipotético*

$$(p \to q) \wedge (q \to r) \vDash (p \to r)$$

nos permite aplicar as regras de transitividade e fundamentar hipóteses consistentemente, ou seja, se r ocorre devido a q se q ocorre devido a p, é trivial que r ocorra devido a p. Para exemplificar o primeiro caso, temos que: "Se (p) eu beber, (q) você dirige. Se você dirige, (r) você não bebe. Eu bebi! Logo, você não bebe".

O *Silogismo Disjuntivo*

$$(p \vee q) \wedge \neg p \vDash q$$

nos permite por sua vez eliminar hipóteses, uma vez que se considera a regra p ou q e se constata não p, consequentemente ocorre q. Por exemplo: "Ou (p) eu trabalho, ou (q) eu descanso. Hoje, eu não trabalho! Logo, hoje descanso".

Dilema: Um dilema normalmente busca desambiguar construções hipotéticas a partir de disjunções entre argumentos base hipótese. O *Dilema Construtivo*

$$(p \to q) \wedge (r \to s) \wedge (p \vee r) \vDash (q \vee s)$$

aplica, ainda que hipoteticamente, as regras de inferência. Assim, se p implica q, e se r implica s, com a ocorrência de p ou r resulta em q ou s. Por exemplo: "Se (p) o time A treinar bem, (q) ele ganha o jogo. Se (r) chover no dia do jogo, (s) o time B leva vantagem. Sabe-se que o time treinou bem e choveu no dia do jogo! Logo, o time A ganhou o jogo ou o time B levou vantagem".

O *Dilema Destrutivo*

$$(p \to q) \wedge (r \to s) \wedge (\neg q \vee \neg s) \vDash (\neg p \vee \neg r)$$

é análogo ao dilema construtivo, no entanto, com premissas negadas nas disjunções. Assim, se p implica q, e se r implica s, com a ocorrência de $\neg p$ ou $\neg r$ resulta em $\neg q$ ou $\neg s$. Exemplo: "Se (p) eu pensar bem, (q) eu encontrarei a resposta; e ainda, se (r) eu deixar a preguiça de lado; (s) obterei êxito. Mas se não encontrei a resposta, ou não obtive êxito, logo ou eu não pensei bem ou eu não deixei a preguiça de lado".

Teorema de Morgan: O teorema de Morgan (bem como na teoria dos conjuntos) é a propriedade de transpor negações de conjunções (ou disjunções) em disjunções (ou conjunções) de negações.

A primeira versão do Teorema de Morgan

$$\neg(p \wedge q) \vDash (\neg p \vee \neg q)$$

diz que a negação de (p e q) tem como consequência (não p ou não q). Para exemplificar: Se é "mentira que Severino é (p) estudante e (q) trabalha." temos que é "verdade que Severino não é estudante ou não trabalha.".

A segunda versão do Teorema de Morgan

$$\neg(p \vee q) \vDash (\neg p \wedge \neg q)$$

diz que a negação de (p ou q) tem como consequência (não p e não q). Por exemplo: Se é "mentira que Severino é (p) baiano ou (q) cearense." temos que é "verdade que Severino não é baiano nem é cearense.".

Se é "mentira que Joana gosta de (p) correr ou (q) andar de bicicleta, mas me chamou de mentiroso por haver dito isso. Logo, Joana gosta de não correr e não andar de bicicleta".

Materialidade: A materialidade tem como objetivo reescrever uma expressão lógica usando um conectivo diferente do original. Por exemplo, a *Equivalência Material*

$$(p \leftrightarrow q) \vDash (p \rightarrow q) \wedge (q \rightarrow p)$$

transforma uma equivalência em uma conjunção de implicações. Exemplo: "(p) Eu irei aí se e somente se (q) você me preparar um jantar" é o mesmo que dizer "Se eu for aí você me prepara um jantar" e "Se você me preparar um jantar, eu irei aí ".

Por outro lado, a *Implicação Material*

$$(p \rightarrow q) \vDash (\neg p \vee q)$$

por exemplo, transforma uma implicação em uma conjunção. Exemplo: "(p) Se você estudar bastante, (q) obterá um certificado" é o mesmo que dizer "Ou você não estudará bastante ou obterá um certificado".

Exercício 26. Prove com a tabela-verdade cada umas das formas básicas de argumentos até aqui apresentados, ou seja, os modus, dilemas, teoremas de Morgan e de materialidade.

Propriedades lógicas: Várias propriedades lógicas são igualmente úteis, mas por conta de escopo desse material, nos limitaremos apenas a citar na continuação:

Associatividade conjuntiva: $p \vee (q \vee r) \vDash (p \vee q) \vee r$

Associatividade disjuntiva: $p \wedge (q \wedge r) \vDash (p \wedge q) \wedge r$

Comutativa conjuntiva: $p \wedge q \vDash q \wedge p$

Comutativa disjuntiva: $p \vee q \vDash q \vee p$

Distributiva conjuntiva: $p \wedge (q \vee r) \vDash (p \wedge q) \vee (p \wedge r)$

Distributiva disjuntiva: $p \vee (q \wedge r) \vDash (p \vee q) \wedge (p \vee r)$

Tautologia conjuntiva: $p \vDash p \wedge p$

Tautologia disjuntiva: $p \vDash p \vee p$

Adição: $p \vDash p \vee q$

Dupla Negação: $\neg\neg p \vDash p$

Contrapositiva: $(p \to q) \vDash (\neg q \to \neg p)$

Composição: $(p \to q) \wedge (p \to r) \vDash p \to (q \wedge r)$

Exercício 27. Construa um exemplo para cada propriedade listada anteriormente.

Lógica proposicional de primeira ordem

Até então, todos os argumentos lógicos são construídos sem levar em consideração o *domínio* do discurso. Sendo assim, muitas vezes as proposições citam elementos sem considerar o conjunto universo no qual estes elementos estão inseridos. Muitas vezes, em situações reais, devemos considerar o domínio onde os elementos citados nas proposições precisamos são declarados. Neste sentido, a *lógica de primeira ordem* surge para dar mais poder de expressividade na elaboração de fórmulas lógicas.

Em lógica de primeira ordem, podemos escolher um elemento qualquer dentro de um conjunto arbitrário através do *quantificador existencial* $\exists$ (existe ao menos um). Assim, dizer que *existe* um elemento dentre o conjunto de cores pode ser expresso como $\exists c \in \{\text{amarelo, azul, branco, verde, vermelho, } \dots\}$. Note que a cor c é arbitrária, tomada dentre o conjunto de cores.

Analogamente, podemos expressar a totalidade dos elementos dentro de um conjunto a partir do *quantificador universal* $\forall$ (para todo). Num outro exemplo, dizer que todos os elementos dentre um conjunto de cores pode ser expresso como $\forall c \in \{\text{amarelo, azul, branco, verde, vermelho, } \dots\}$. Neste caso, c assume a posição de todas as cores possíveis dentro do conjunto.

Na lógica de primeira ordem, a teoria dos conjuntos entra como uma ferramenta essencial na delimitação do que chamamos de conjunto universo.

Exemplo 15. Num exemplo prático, considere um conjunto de pessoas P. Como listar todas as pessoas casadas desse grupo? Neste caso, a proposição pode ser descrita como

$$\{\forall p \in P : p \text{ é casado.}\}$$

Exemplo 16. Como saber se podemos chamar um grupo de pessoas G de "eles" ou "elas"? Pela língua portuguesa, podemos averiguar se pelo menos uma pessoa do grupo é do gênero masculino. Ou seja, basta satisfazer a seguinte proposição:

$$\{\exists p \in G : p \text{ é do sexo masculino.}\}$$

Com o uso da lógica de primeira ordem, podemos construir argumentações mais robustas e completas, levando assim, mais poder de abstração ao processo de composição das bases de conhecimento. Desse exemplo anterior, podemos derivar a expressão

$$\{\exists p \in G : p \text{ é do sexo masculino.}\} \rightarrow \text{“chame por eles”},$$

bem como a expressão

$$\neg\{\exists p \in G : p \text{ é do sexo masculino.}\} \rightarrow \text{“chame por elas”}.$$

Exemplo 17. “Roberto se comprometeu a pagar as mensalidades de seu filho Joãozinho”, se e somente se, “ele nunca tirasse alguma nota inferior a 8.0 na faculdade”. Em linguagem formal,

$$p \leftrightarrow \neg\{\exists n \in \text{Notas} : n < 8.0\},$$

onde p é a sentença do compromisso de Roberto e “Notas” é o conjunto de todas as notas de Joãozinho. É comum parametrizar sentenças que incluem um elemento quantificado da seguinte maneira: $q(n) = n < 8.0$. Assim, conseguimos tornar paramétrica a fórmula que descreve o argumento.

Observe que o segundo argumento pode ser expresso alternativamente da seguinte forma: “Toda nota de Joãozinho da faculdade deve ser superior ou igual a 8.0”, ou seja,

$$p \leftrightarrow \{\forall n \in \text{Notas} : n \geq 8.0\}.$$

No caso do exemplo anterior, observe que a proposição $n \geq 8.0$ é a negação da proposição $q(n) = n < 8.0$. Ou seja, $\neg q(n) = n \geq 8.0$. Intuitivamente, pelo exemplo anterior exploramos uma importante propriedade dos quantificadores, a negação. Dado um conjunto arbitrário S, um elemento n e uma fórmula q que inclui n como parâmetro, é válido que:

$$\neg\{\exists n \in \text{S} : q(n)\} \leftrightarrow \{\forall n \in \text{S} : \neg q(n)\},$$

assim como é válido que

$$\neg\{\forall n \in \text{S} : q(n)\} \leftrightarrow \{\exists n \in \text{S} : \neg q(n)\}.$$

Se aplicarmos essa propriedade ao exemplo anterior, podemos derivar duas expressões equivalentes:

$$\{\exists p \in G : p \text{ é do sexo masculino.}\} \rightarrow \text{“chame por eles”},$$

assim como

$$\{\forall p \in G : p \text{ não é do sexo masculino.}\} \rightarrow \text{“chame por elas”}.$$

Bases da álgebra Booleana

A álgebra Booleana é um tipo de formalismo que melhor expressa a lógica computacional, por trabalhar apenas com zeros e uns. No restante, os conectivos da lógica proposicional se tornam operadores binários $\vee$ (o “ou” lógico, denotado por $+$) e $\wedge$ (o “e” lógico, denotado por $*$), uma operação unária $\neg$ (o “não”

lógico, denotado por $\sim$), e duas constantes 0 (chamado de “zero” ou "falso", também denotada por F) e 1 (chamado de “um” ou “verdadeiro”, também denotada por V), e satisfazendo os seguintes axiomas, para quaisquer $a, b, c \in X$:

Identidade	$0 = 0, 1 = 1$	$0 \neq 1$
Negação	$\neg(a \vee b) = \neg a \wedge \neg b$	$\neg(a \wedge b) = \neg a \vee \neg b$
Propriedades Associativas	$(a \vee b) \vee c = a \vee (b \vee c)$	$(a \wedge b) \wedge c = a \wedge (b \wedge c)$
Propriedades Comutativas	$a \vee b = b \vee a$	$a \wedge b = b \wedge a$
Propriedades Absortivas	$a \wedge (a \vee b) = a$	$a \vee (a \wedge b) = a$
Propriedades Distributivas	$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$	$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$
Elementos Neutros	$a \vee 0 = a$	$a \wedge 1 = a$
Elementos Complementares	$a \vee \neg a = 1$	$a \wedge \neg a = 0$

Tabela 4.7: Propriedades elementares da álgebra Booleana

A vantagem de se usar álgebra Booleana é o benefício de se usar todo o formalismo algébrico para avaliar expressões por um computador digital.

Exemplo 18. Dada a seguinte expressão Booleana $a \wedge b \vee c$, considerando que $a = 1$, $b = 0$ e $c = 1$, temos que o resultado da avaliação da mesma é $1 \wedge 0 \vee 1 = 0 \vee 1 = 1$, ou seja, verdadeira.

Se perguntássemos às pessoas quanto é a metade de dois mais dois, a maioria reponderá prontamente 2. Resposta errada! Pois ao aplicarmos as regras de precedência das operações matemáticas constatamos que, por ordem de prioridade, a multiplicação e a divisão devem ser calculadas antes de somas e subtrações. Sendo assim, a metade de dois mais dois é 3, pois $2/2 + 2 = 1 + 2 = 3$. Na álgebra Booleana vale o mesmo argumento, ou seja, a regra de precedência consiste em:

1. Avaliar expressões entre parênteses;
2. Avaliar expressões de negação ($\neg$);
3. Avaliar expressões conjuntivas ($\vee$);
4. Avaliar expressões disjuntivas ($\wedge$).

Exemplo 19. Dada a seguinte expressão Booleana $a \vee b \wedge c$, considerando que $a = 1$, $b = 0$ e $c = 1$, temos que o resultado da avaliação da mesma é $1 \vee 0 \wedge 1 = 1 \vee 0 = 1$, ou seja, verdadeira. Note que neste caso, aplicamos a regra de precedência de $\vee$ sobre $\wedge$. A dada expressão pode ser escrita como $a * b + c$.

Exercício 28. Encontre o valor das seguintes expressões Booleanas considerando que $a = 1$, $b = 1$, $c = 0$ e $d = 1$:

(1) $a \vee (b \wedge c)$

(2) $a \vee (\neg b \wedge c \vee d)$

(3) $\neg a \wedge \neg(b \wedge c \vee d)$

(4) $a \wedge \neg(\neg b \wedge c \vee d)$

Enquanto percebe-se um claro movimento de cientistas e engenheiros computacionais na direção de *ensinar*

ao computador a linguagem humana, nós da Looplex acreditamos na via oposta, isto é, *ensinar ao ser humano a linguagem computacional*. Isso porque entendemos que no direito, o mínimo esperado é que a máquina acerte 100% em suas decisões jurídicas. Por questões éticas e outras discussões relacionadas, o engenheiro jurídico responde pelas consequências que seu código pode acarretar, assim como em construções civis. Logo, enxergue a linguagem de programação como uma nova língua a se aprender, assim como o inglês, francês, ou qualquer outra língua estrangeira.

4.3 Fundamentação algorítmica

Um *algoritmo*, ou *programa*, é uma linguagem na qual se apresenta uma sequência de regras, declarações e operações que, aplicada sobre um conjunto de dados, nos permite solucionar problemas.

A base de um algoritmo são os instruções à máquina de modo a executar tarefas específicas. No caso de automação de documentos, são linhas de comandos que dirigem o computador em tarefas de composição de documentos e outros procedimentos afins.

Existem paradigmas relacionados ao modo em que o programador constrói seus raciocínios, e os estruturam de modo a resolver o problema objetivo.

Paradigmas clássicos de programação

De um modo geral, podemos destacar três paradigmas de programação [36] clássicos, bem como uma breve reflexão sobre o modo de pensar de cada um deles.

Programação estruturada: Dentre os paradigmas de programação, a programação estruturada é sem dúvidas o paradigma mais adotado entre todos os outros. É muito difundido em disciplinas de computação e implementado por linguagens de programação como Ada, Basic, C, Cobol e Pascal fazem parte dessa forma de estruturar um programa.

Basicamente, a programação estruturada, e especificamente a programação sequencial, gerencia o fluxo de execução de um programa, computando uma instrução de cada vez. Nesse paradigma, qualquer algoritmo é reduzido a três estruturas fundamentais:

1. Estruturas de controle: também chamadas de estruturas sequenciais;
2. Estruturas de decisão: também chamadas de estruturas condicionais ou de seleção;
3. Estruturas de repetição: também chamadas de estruturas de iteração ou laços.

As estruturas de controle são basicamente comandos que são interpretados pela máquina de modo sequencial. Obviamente, cada instrução dentro do fluxo de execução deve ser compreendida pela máquina. Se por exemplo, a máquina não entender o comando “Ligue para o Fulaninho”, ela nada poderá fazer a não ser responder: “Não entendi”.

As estruturas de decisão por sua vez funcionam como desvios do fluxo de execução dependendo de condições específicas. É como se em uma árvore de possibilidades nos deparamos com uma bifurcação. Tal bifurcação é definida por uma estrutura condicional.

Por fim, as estruturas de repetição são desvios específicos do fluxo de execução, retomando um estado anterior na sequência de comandos. É basicamente o poder de repetir uma sequência de operações dentro do fluxo de execução do programa até que (ou enquanto) se cumpra uma determinada condição.

A partir dessas três estruturas algorítmicas, conseguimos construir programas de uma complexidade estrutural imensurável. Foi justamente preocupado com a falta de inteligibilidade do código produzido pelos próprios seres humanos, que novos paradigmas de programação foram propostos, para que haja um maior grau de entendimento para fins de manutenção e expansão dos algoritmos. Um dos paradigmas com grande destaque é a programação orientada a objetos.

Programação orientada a objetos: Esse paradigma apresenta um conjunto de métodos, técnicas e padrões de software que auxiliam o programador a modelar, projetar e programar sistemas robustos de modo mais natural. Dentre as linguagens de programação que implementam a programação orientada a objetos se destacam Smalltalk, Java, C++, C#, Objective-C, VB.NET e Object Pascal.

O fluxo de execução em programação orientada a objetos administra a interação entre *objetos*. Cada um desses objetos são vistos como pequenos programas com código próprio. Esses objetos reúnem um conjunto de estados (propriedades e atributos internos) e um conjunto de comportamentos (métodos) de modo que trocam mensagens entre si, desenvolvendo a dinâmica da programação em si mesmo. Entretanto, o fluxo de execução ainda existe, e é controlado por algum algoritmo principal (muito embora possam haver variados fluxos independentes e em paralelo, mas ainda assim, cada fluxo independente é um programa rodado em paralelo).

Outro aspecto interessante na programação orientada a objetos é a definição de classes de objetos. Literalmente, classificamos os objetos, e os agrupamos em classes comuns. Por exemplo, se tomássemos o seguinte conjunto {carro, cachorro, garfo, gato, caminhão, pato, papagaio, prato}, poderíamos criar as seguintes classes: “Seres Vivos”, “Seres Inanimados”, “Mamíferos”, “Aves”, “Meios de Transporte”, “Utensílios Domésticos”. O mais importante em critérios de classificação é garantir a máxima cobertura dos objetos por classes que se complementam. Ou seja, não é conveniente classificarmos esses elementos em três classes: “Mamíferos”, “Utensílios Domésticos” e “Outros”, visto que não existe nenhuma relação afim entre “pato” e “carro”, embora pertençam à classe “Outros”.

Exercício 29. A fim de desenvolver a habilidade cognitiva do leitor, categorize apropriadamente os seguintes elementos do conjunto {relativamente incapaz, casado, pessoa natural, EIRELI, convivente estável, pessoa jurídica, capaz, sociedade limitada, sociedade anônima, solteiro, ente não-personificado, autarquia, parcialmente incapaz, divorciado}

Uma dificuldade encontrada muitas vezes na classificação é a sobreposição de classes, e às vezes até mesmo casos de subconjuntos de classes: subclasse. Esse conceito é conhecido como generalização, por exemplo, uma pessoa capaz é uma pessoa natural, bem como uma pessoa parcialmente incapaz. Ou seja, a classe natural é uma generalização de pessoa capaz e parcialmente incapaz. A recíproca é dita ser uma especialização, ou seja, pessoa capaz é uma especialização de uma pessoa natural. Esses conceitos são fundamentais para o desenvolvimento de um bom projeto de algoritmos.

Vale salientar que a programação orientada a objetos enfatiza o estado de um objeto ao longo do tempo (dentro do fluxo de execução). Existem outros paradigmas que não adotam esse enfoque. Um bom exemplo disso é a programação funcional.

Programação funcional: Nesse paradigma, os algoritmos adotam funções como elementos básicos para expressar comandos e procedimentos que a máquina deve executar. São tidas como linguagens de programação funcional LISP, Haskell e Scala. Pode-se pensar na programação funcional como simplesmente avaliação de expressões sucessivas. O paradigma de programação funcional é o mais antigo que se conhece (desde 1955 com a linguagem IPL).

O programador define uma função para resolver um problema e passa essa função para o computador. Tal função pode envolver várias outras funções em sua definição. O Computador funciona então como uma calculadora que avalia expressões escritas pelo programador através de simplificações até chegar a uma forma normal. Tal forma normal é então executada pela máquina.

Em linguagens estruturadas, o foco está no estado do programa. Sendo assim, containers de dados devem ser declarados ao longo do programa a fim de armazenar o estado de cada uma dessas informações. Por exemplo, se desejássemos criar um programa para enviar cartões natalinos por email, para cada etiqueta precisamos armazenar o nome do remetente e o nome do destinatário. Essas informações são guardadas durante o fluxo de execução para que no momento da impressão, esses valores sejam retomados.

Em linguagens funcionais, os dados não são declarados mas sim, passados como parâmetros (argumentos) para essas funções. Ou seja, se queremos pedir para um robô enviar cartões natalinos de João para Maria via programação funcional, o código seria algo do tipo

envie(“cartão de natal”).por(“email”).de(“João”).para(“Maria”),

onde o ponto é um delimitador que sinaliza concatenação de funções a serem executadas consecutivamente. Observe que nesse caso, a propriedade comutativa pode ser aplicada ao exemplo sem modificar o sentido do comando, ou seja,

envie(“cartão de natal”).de(“João”).para(“Maria”).por(“email”).

Novos processadores com múltiplos núcleos estão sendo desenvolvidos hoje em dia. No entanto, o sucesso desse panorama só é necessário com a escrita de programas que funcionem de forma paralela. Como a programação estruturada é regida por fluxo de execução, a coordenação desses fluxos torna-se um problema intransponível. Daí surge o apelo de se usar programação funcional, pois é muito mais fácil escrever um código concorrente sem qualquer preocupação com o estado dos dados armazenados.

Embora alguns não percebam, os benefícios da programação funcional já estão sendo incorporados às linguagens estruturadas de modo natural, como por exemplo, o uso de funções Lambda. A notação Lambda pode ser visto como uma linguagem de programação abstrata em que funções podem ser combinadas para formar outras funções.Tal notação é usada para especificar funções e definição de funções, e por serem fáceis de implementar, são muito bem difundidas em diversas linguagens de programação (Java, C# ou Javascript por exemplo).

Segue um exemplo de notação Lambda a título de curiosidade da definição de uma função que calcula a área de um retângulo de base x e altura y. Para quem matou essa aula, não tem problema, aqui vai a cola. A

área de desse retângulo é $x * y$. Sendo assim, a função f que calcula essa área é descrita como:

$$f : (x, y) \mapsto x * y.$$

Exercício 30. Experimente você criar uma função Lambda para calcular o perímetro de um retângulo de base x e altura y. Lembrando que o perímetro de um retângulo é a soma do comprimento de todos os seus lados.

Outros paradigmas de programação surgem e enriquecem o modo de pensar do programador. A programação orientada a eventos [37], por exemplo, reconfigura o fluxo de execução de modo que a execução é guiada por interrupções e troca de mensagens externas, os então denominados eventos. No caso da programação reativa [38] o programa lida com fluxo de dados assíncronos, conhecido como *stream*. A programação orientada a aspectos [39], por sua vez, preconiza a organização estruturada do código de modo a segmentar o código de acordo com a sua relevância para a aplicação.

Não é interesse desse material esgotar todos os paradigmas de computação. Apenas cabe elucidar por quais caminhos "programáticos" o leitor pode se dirigir, de acordo com suas habilidades. Existe uma corrente que defende o conceito de linguagem multiparadigma, de modo a unificar conceitos chaves de diversos paradigmas de programação. Tal conceito nos permite compor uma caixa de ferramentas da qual o programador é capaz de aplicar variados conceitos e estilos livremente. No meio desse cenário foi construído o *Lawtex*.

Afinal de contas, o que é um programa?

Um programa, ou um algoritmo, é uma sequência de instruções que pode ser executada por uma máquina. Para explicar o que é um programa de modo intuitivo, vamos a um exemplo prático. Imagine que desejamos realizar uma tarefa cotidiana: "como fazer arroz?".

Em uma busca rápida pelo Google, peguei a seguinte receita (me perdoe qualquer equívoco culinário, cozinhar não é o forte deste que vos escreve).

1. Refogue o alho com a cebola no azeite.
2. Adicione o arroz na panela.
3. Deixe o arroz fritar por cerca de 30 segundos, pois assim ele ficará mais soltinho.
4. Adicione a água fervente e o sal.
5. Abaixe o fogo e deixe cozinhar até a água ter quase secado.
6. Tampe a panela e aguarde cerca de 20 minutos antes de servir.

A fim de facilitar o raciocínio, desconsideramos as quantidades de cada ingrediente. Partindo do princípio que o programa é executado por uma máquina digital, podemos considerar nossa máquina um cozinheiro do sexo masculino (uma vez que tradicionalmente se assume que homens conseguem executar apenas uma instrução por vez). Está claro o fluxo de execução para um ser-humano. Um ser humano entende perfeitamente que a instrução "Abaixe o fogo e deixe cozinhar até a água ter quase secado." implica em duas ações. Na primeira

ele irá regular o fogo até que o marcador indique fogo baixo. Então, num processo assistido, ele esperará até que a água dentro da panela quase se seque, ou seja, aguardará pelo evento "água ter quase secado".

Note ainda que existem informações que apenas explicam/justificam ações correntes ou não trazem nenhum comando explícito, "pois assim ele ficará mais soltinho" e "antes de servir". Todos esses termos são irrelevantes dentro do algoritmo, pois trazem explicações inúteis para o robô, bem como a citação de eventos futuros sem qualquer formalização específica para o computador. Sendo assim, é importante que no processo de construção do algoritmo se elimine qualquer comando desnecessário ou incompreensível pela máquina. Conclusão, o computador é muito mais burro do que imaginamos.

O computador possui um recurso bem limitado de instruções. Considere que ele apenas obedeça ordens cegamente, sem analisar qualquer estrutura semântica de uma sentença mais complexa, como uma "oração subordinada adverbial temporal" como nós seres humanos fazemos (eu sei... essa desenterrei do ensino médio :-|). Neste caso, vamos reescrever a receita de um modo mais compreensível para máquinas obtusas, usando apenas sentenças imperativas. Para trazer um pouco mais de realidade, vou considerar um acréscimo a receita para o caso de não haver cebola. Eu imagino que se você não tiver cebola, basta se contentar com arroz, alho e sal.

1. **Pique** o alho.
2. **Caso** tenha cebola.
3. **Pique** a cebola.
4. **Escolha** uma panela.
5. **Acenda** uma boca no fogão.
6. **Regule** o fogo para o nível "fogo alto".
7. **Leve** a panela ao fogão.
8. **Adicione** azeite à panela.
9. **Caso** tenha cebola.
10. **Adicione** a cebola picada à panela.
11. **Adicione** o alho picado à panela.
12. **Enquanto** os ingredientes **não** estiverem dourados
13. **Aguarde** 10 segundos.
14. **Adicione** o arroz à panela.
15. **Aguarde** 30 segundos.
16. **Adicione** a água fervente.
17. **Adicione** o sal.
18. **Regule** o fogo para o nível "fogo baixo".
19. **Enquanto** o nível a água **não** estiver baixo
20. **Aguarde** 1 minuto.

21. **Tampe** a panela.
22. **Aguarde** 20 minutos.

Um robô que saiba avaliar expressões condicionais ("**Caso** tenha cebola"), souber avaliar eventos repetitivos (**"Enquanto** o nível a água **não** estiver baixo") e executar comandos como picar ingredientes, escolher panelas, ligar o fogão, regular o fogo, levar panelas ao fogão, adicionar ingredientes à panela, aguardar por eventos específicos e tampar uma panela, conseguiria fazer um arroz sem dificuldades. Um algoritmo, ou programa, é em essência isto: uma sequência de condicionais, repetições e comandos executáveis por uma máquina.

Fluxo de declaração

Veremos o uso da palavra declaração várias vezes. Ela sempre aparece em oposição à operação. Todas as avaliações, lógicas, comparações etc. que programamos se baseiam em alguma informação fornecida pelo usuário ou elaborada pelo sistema com base em informações fornecidas pelo usuário. Essas informações são a caixa de ferramentas. Assim, temos que declarar, ou seja, definir quais informações queremos utilizar e qual é o tipo dessas informações. Do contrário, pediremos para o sistema determinar se o autor é pessoa física ou jurídica e o sistema nem encontrará um autor para fazer a avaliação! As informações precisam existir antes de serem utilizadas.

Operandos: Matematicamente, uma operação qualquer é composta por um ou mais operandos concatenados por operadores. Um operando nesse sentido corresponde a uma unidade informacional dentro da operação na qual ele está inserido. É por meio dos operandos que os programas conseguem armazenar os valores, dados e conceitos atribuídos pelo usuário do programa. No caso, o usuário executa o programa e o programa requer dele que valores específicos sejam atribuídos aos operandos internamente declarados.

Exemplo 20. Na operação matemática $5+7$ temos um operador de adição ($+$) conectando dois operandos constantes 5 e 7.

Exemplo 21. Abstratamente, poderíamos expressar a operação "soma de dois números inteiros x e y" da seguinte forma: $x + y$. Observe que nesse caso, o valor de x e de y variam, por isso são chamadas variáveis.

Exemplo 22. Analogamente, poderíamos somar números em várias dimensões, como por exemplo

$$\langle 1, 2, 3\rangle + \langle 4, 5, 6\rangle = \langle 5, 7, 9\rangle.$$

Neste caso, consideramos cada operando $\vec{v} = \langle x, y, z, \ldots\rangle$ como uma coleção de números, e não uma representação atômica dos números naturais, como no exemplo anterior. Essa coleção homogênea de números, ou seja, somente de números, denominamos vetor numérico.

Em programação o mesmo conceito é válido. Um operando serve não somente para armazenar números, como nos exemplos anteriores, mas qualquer tipo de dados, sejam atômicos: informacionalmente indivisíveis tais como números, textos, datas e horários; coleções homogêneas: como no caso de vetores de um determinado tipo específico; ou ainda coleções heterogêneas: dados estruturados pela composição de outros operandos.

Tipos: Quando declaramos variáveis ou conjuntos definimos seu tipo. Como veremos, há tipos primitivos e tipos que chamaremos de objetos ou estruturas. Os tipos primitivos são textos, números inteiros, números reais e booleanos.

A maioria das linguagens de programação são tipadas. O único tipo que não é autoexplicativo é o booleano, que é o nome que se dá a variáveis que comportam "verdadeiro" ou "falso". Um preço pode ser expresso por uma variável que é um número real. Número de filhos pode ser expresso por uma variável que é um número inteiro. Nome pode ser expresso por uma variável que é um texto. Por fim, podemos expressar se uma parte no processo juntou procuração ou não por uma variável que é um booleano, ou seja, ou a parte juntou a procuração ou não juntou.

Por fim, um objeto é um tipo complexo que agrega diversas informações. Uma pessoa é um número, um texto ou um booleano? Nenhum deles. Ela é um agregado. Uma pessoa tem um nome, que é um texto, um telefone, que é um número, bem como estado civil, endereço etc. Assim um objeto é um tipo complexo criado a partir dos demais tipos.

Conjuntos e sequências: Assim como temos variáveis de um tipo podemos ter conjuntos de um tipo. A parte autora em um processo pode ser uma variável do tipo pessoa, mas se quisermos permitir que o usuário lide com casos em que há mais de uma parte autora precisaremos declarar um conjunto de partes autoras, que pode ter um ou mais elementos.

Quando usamos conjuntos precisamos ficar atentos para usarmos notações de conjuntos. Um exemplo disso é fazer avaliações para cada elemento que pertencer ao conjunto, ao invés de fazer só uma avaliação.

Fluxo de operação

Conforme comentado na Seção 4.3, um algoritmo estruturado é composto por uma sequência de instruções, as quais são executados sequencialmente.

Existem vários tipos específicos de instruções

Estruturas de seleção: Em certas linguagens de programação, desvios condicionais podem ser expressos por meio de múltiplos casos. No entanto, uma estrutura condicional pode ser representada por diretivas do tipo

SE <expressão Booleana> **ENTÃO**

{instruções executadas caso a <expressão Booleana> seja avaliada como VERDADEIRA}

SENÃO

{instruções executadas caso a <expressão Booleana> seja avaliada como FALSA}

FIM-SE

Embora seja intuitivo por si só o que o código anterior realiza, vale salientar que as instruções executadas dentro do escopo dos blocos condicionais podem assumir qualquer elemento

Estruturas de repetição: De um modo geral, uma estrutura condicional pode ser representada por comandos do tipo:

ENQUANTO <expressão Booleana> **FAÇA**

{instruções executadas enquanto <expressão Booleana> seja avaliada como VERDADEIRA}

FIM-ENQUANTO

Em certos casos queremos explorar os elementos de um vetor individualmente. Em programação chamamos isso de iterar o vetor. Assim, dado um vetor qualquer, a seguinte estrutura é capaz de pegar elemento a elemento e executá-lo dentro do bloco de instruções correspondente:

PARA-CADA <elemento> DE <nome do vetor> **FAÇA**

{instruções executadas sobre cada <elemento>}

FIM-PARA-CADA

Operações: Operação é o uso de uma informação. O mesmo vale para avaliações, comparações, cálculos etc. As operações, em diversas linguagens, são feitas em um espaço separado do código, não se misturando com as declarações. Uma das operações mais comuns dentro das linguagens estruturadas é a atribuição. A atribuição altera o estado interno dos operandos de modo que ao longo do fluxo de execução, esses operando vão assumindo valores diferentes.

Por exemplo, se realizarmos a seguinte atribuição

$$x \longleftarrow 1$$

num determinado trecho de código, imediatamente depois, toda vez que nos referirmos a x, estamos na verdade falando do valor atribuído a ele, ou seja 1. Se depois disso realizarmos a seguinte atribuição

$$x \longleftarrow x + 1,$$

o valor de x passa imediatamente a ser 2 após essa instrução.

Comandos: Em suma, dentro das linguagens de programação, um comando é uma operação específica que tem uma palavra que lhe é reservada para fins de identificação e execução. São diretivas que expressam uma ação a ser tomada pelo computador. Quando imprimimos uma variável, por exemplo, estamos dando um comando ao computador para que exiba a informação que ela contém. Normalmente, um comando apresenta a forma direta expressa no imperativo:

FAÇA <algo>

Em geral, o verbo **FAÇA** representa um conjunto de comandos, dos quais se destacam: **LEIA**, **ESCREVA**, **CALCULE**, entre outros.

Funções e métodos: Em programação, funções existem como na matemática. Essas funções são operações que podem receber um argumento como entrada (*input*) e devolver um valor de saída (*output*). Quando uma função não devolve nenhum valor, costumamos chamá-la de procedimento. Tanto na matemática quanto na programação elas comportam um conjunto de instruções. Enquanto na matemática a função impõe que se faça uma série de contas com o input, na engenharia jurídica podemos, por exemplo, receber um conjunto de partes devedoras, selecionar as que são casadas e redigir uma cláusula discorrendo sobre outorga uxória, mencionando cada uma delas. Uma função em programação é expressa de modo similar como se expressa na matemática, isto é, parametrizada com operandos dentro de parênteses (argumentos da função):

EXECUTE(<operando$_1$>, <operando$_2$>)

Em geral, o verbo **EXECUTE** leva o nome da função específica, como por exemplo, **MÁXIMO_ENTRE**, **MÍNIMO_ENTRE**, **ORDENE**, etc.

Exemplo 23. A partir das instruções anteriormente apresentadas, possuímos a aptidão para desenvolver algoritmos em pseudocódigo para diversas tarefas básicas. Neste exemplo, vamos ler dois números e escrever a relação entre eles:

LEIA n_1

LEIA n_2

SE $n_1 = n_2$ **ENTÃO**

  ESCREVA n_1 & " é igual a " & n_2

SENÃO

  SE $n_1 > n_2$ **ENTÃO**

    ESCREVA n_1 & " é maior do que " & n_2

  SENÃO

    ESCREVA n_1 & " é menor do que " & n_2

  FIM-SE

FIM-SE

É padrão representar textos delimitados por aspas duplas. Assim, evitamos confusões na hora de decidir se um espaço em branco será impresso ou não, por exemplo. O sinal '&' é um operador de concatenação de texto. No exemplo anterior, ocorreu uma estrutura condicional *aninhada* dentro da outra. É muito comum em programação aninharmos condicionais dentro condicionais, laços dentro de laços, e assim, livremente, se compõe o algoritmo.

Exercício 31. Dentro da automação de documentos jurídicos, a qualificação das partes de um contrato é uma das tarefas básicas mais importantes. Escreva um algoritmo para **LER** as seguintes informações de um determinado cliente: **tipo_de_pessoa** (natural ou jurídica), **nome**, **endereço**, **nacionalidade**; se for pessoa física também pedir **profissão**, **estado_civil**, **rg** e **cpf**; se for pessoa jurídica pedir informações tal como **cnpj**, **nome_do_representante_legal**, **rg** e **cpf**. Usando essas informações pedidas, o algoritmo deve também **ESCREVER** uma qualificação básica.

5 Codificação com Lawtex

Aqui retomamos os conceitos de programação sugeridos anteriormente, falamos que o Lawtex é uma linguagem voltada para profissionais do direito e indicamos o que é sintaxe e que apresentaremos a sintaxe do Lawtex.

5.1 Meu primeiro programa: "*Hello Judge*"

Abaixo, para quebrar um pouco o gelo, vamos codificar o nosso primeiro programa em Lawtex.

```
template[TEMP_HelloJudge] {
    metainfo {
        language = "pt_BR"
    }
    body {
        operations {
            print "Olá juíz!"
        }
    }
}
```

O código acima não faz nada de especial. Toda a estrutura do código, a forma de escrever, as restrições de chaves e nomes dos comandos definem um vocabulário específico. Esse vocabulário que compõe a linguagem Lawtex é regido por regras sintáticas para a perfeita compreensão do algoritmo pela máquina. Não apresenta nenhuma ação ao usuário, isto é, não gera nenhum *card*, e ainda por cima imprime "Olá juiz!". No caso, todo o template em Lawtex deve ter, no mínimo essa estrutura, ou seja, a diretiva "template" com um identificador iniciado por letra maiúscula entre colchetes, e um bloco de propriedades delimitados por chaves. Cada propriedade por sua vez, segue suas particularidades sintáticas, as quais serão melhor exploradas na sequência.

5.2 Definições de Sintaxe

Sintaxe é um aspecto necessário, porém menos importante na engenharia jurídica. A sintaxe corresponde ao formalismo da linguagem em que se escreve os códigos interpretados pela máquina. Cada linguagem de programação tem sua própria sintaxe: suas regras gramaticais, palavras reservadas, formatos de expressar declarações e operações, entre outras.

A importância da sintaxe é operacional, ou seja, você não consegue programar sem sintaxe, contudo a lógica

independe da sintaxe. Na verdade, se este material for bem-sucedido, você aprenderá lógica de programação, aprenderá a formalizar e estruturar a lógica jurídica em declarações, condições, dependências, iterações, etc. Tais estruturas são clássicas em programação estruturada. Dominando esse raciocínio você pode fazê-lo em qualquer linguagem de programação existente, respeitadas as particularidades de cada uma.

Dentre outros critérios, a linguagem **Lawtex** foi escolhida como linguagem de codificação deste material, particularmente pelas facilidades em que se definem os templates e componentes de um template, visto que foi uma linguagem desenvolvida especialmente para profissionais do meio jurídico. Além disso, o feedback do código produzido é imediato, isto é, existe uma plataforma que responsivamente interpreta o código dinamicamente.

Talvez nem toda linguagem dispõe de propriedades como "help" ou "request", mas se você souber, por exemplo, o que é iterar um vetor de objetos, você poderá consultar a sintaxe necessária para fazê-lo em Lawtex nesse material ou na internet para qualquer outra linguagem. Depois de aprender a iterar sobre vetores de objetos com o foreach em Lawtex procure, por exemplo, "How to iterate in Python" no Google e compare a sintaxe com o Python. Você verá que a lógica transcende a implementação.

A tecnologia muda muito rápido, mas seus fundamentos não. Fazendo uma analogia com as línguas, praticamente todas as ideias que podem ser expressas em francês podem ser expressas em inglês, respeitadas algumas poucas particularidades. Tenha isso em mente.

Aprenda a sintaxe, mas sempre pressupondo que seu domínio vem com a prática e que a consulta é sempre possível. Mantenha seu foco na lógica que aprendeu no começo do material. Apesar da relatividade da importância da sintaxe, ela é imprescindível para programar. Lembre-se: a sintaxe é a "fórmula", o formato pelo qual pedimos para o computador executar uma tarefa. Quando redigimos no código a linha

print <dataDeAssinatura>,

estamos pedindo ao sistema que escreva no documento a data de assinatura preenchida ao usuário. Considere o trecho de código abaixo:

```
<valorTotal>.ask(),
<numeroDeParcelas>.ask(),
if (<numeroDeParcelas>.isNotEmpty() AND <valorTotal>.isNotEmpty()) {
    <valorDaParcela> = <valorTotal> / <numeroDeParcelas>,
    print "\p O pagamento será realizado em "& <numeroDeParcelas>
    print "parcelas de R$"& <valorDaParcela> & ".\n\b",
}
```

A tradução desse código para português seria:

"Pergunte ao usuário o valor total do produto",

"Pergunte o número de parcelas",

"Caso o usuário responder essas informações pedidas, ou seja, não forem mais vazias"

"Calcule o valor da parcela, dividindo o valor total pelo número de parcelas",

"Imprima um parágrafo com essa informação: primeiro discrimine o número de parcelas"

"Descreva também os valores delas e por fim, feche o parágrafo e pule uma linha"

Agora releia as linhas de código e compare-as com a versão em português: você verá que elas parecem bem menos intimidadoras.

Dessa forma, a sintaxe é a língua falada pelo computador. O computador é menos sagaz com contexto e erros de digitação que seres humanos. Se o código não for absolutamente perfeito ele não será compreendido e o computador retornará um erro. Esses "erros de digitação" são chamados de erros de sintaxe. Você ***sempre*** cometerá erros de sintaxe. É comum na vida de todo programador e, portanto, de toda engenheira ou engenheiro jurídico. Por isso é interessante validar o código conforme você o redige para não ter que resolver todos os erros de uma só vez. Não se preocupe tanto com a *depuração* (busca minuciosa pelo erro) do código antes de subi-lo no sistema. O sistema informa quando há um erro (mas não informa erros de lógica jurídica).

Sugerimos uma ferramenta de desenvolvimento para codificação: o **Sublime Text 3**. É uma ferramenta leve e muito boa para edição de código. Basta fazer o download no site `https://www.sublimetext.com` para obter uma versão gratuita. Um bom recurso do Sublime é o uso de *snippets*. Eles são pequenos modelos que podem ser acionados no editor de texto que usamos para programar. Eles serão a fonte de consulta "abstrata" da sintaxe. Justamente porque existem os *snippets* optamos, por questões didáticas, procuramos usar exemplos concretos nesse material. Basta escrever "stru" que o editor vai sugerir "struct". Aperte enter e você terá automaticamente um exemplo da sintaxe com lacunas, só sendo necessário preencher o que realmente interessa.

Outra coisa importante é a indentação ou recuo. Nos exemplos de sintaxe você vai reparar que há sempre uma serie de tabulações, recuos inseridos antes de determinados trechos de código. Isso é para identificar em que contexto se inserem. Se estou declarando uma variável e abro chaves para declarar suas propriedades devo dar um "tab" antes de inserir cada uma delas. Isso porque as propriedades de uma variável estão em nível hierárquico ou de profundidade diferente das demais variáveis. Fica muito confuso deixar informações diferentes no mesmo recuo. Além disso, com o recuo correto você pode minimizar trechos internos de código para ter uma visão geral do código. Isso ficará mais claro com o tempo.

Os erros de sintaxe não são os únicos erros. Há erros mais complexos. Imagine que, por acidente, no lugar de dividir o valor total pelo número de parcelas você divida o valor total pela parte devedora. Parte devedora é uma variável que não é um número. Ela é um sujeito, com nome, CPF, endereço etc. O sistema só consegue fazer contas com números. Não faz sentido fazer contas com objetos complexos com diversas informações. Da mesma forma não é possível fazer uma conta com textos. Diante desse erro o sistema acusará uma incompatibilidade na operação, ou seja, você tentou realizar uma operação que não é adequada ao tipo de informação fornecida. Esses erros podem ser evitados, mas isso exige atenção e compreensão do que se está fazendo.

Outro erro comum nesse sentido é pedir uma informação não indexada a um vetor (ver Seção 5.4). O que isso quer dizer? Imagine que você tem um conjunto de partes garantidoras. Essa sequência de garantidores é expresso em Lawtex por um vetor. Como veremos a sintaxe dele é "|garantidores|". Imagine agora que eu escreva a linha

print |garantidores.estadoCivil|,

com o objetivo de imprimir o estado civil de cada garantidor numa cláusula de representações e garantias. Estado civil não é uma característica da sequência de garantidores, mas sim de cada um dos seus elementos. Cada garantidor tem seu próprio estado civil. Logo é preciso indexar, ou seja, identificar o estado civil de

qual dos garantidores se quer. O sistema acusará um erro sem indexação. Nesse sentido se eu inserir a linha

print |garantidores{0}.estadoCivil|,

aí sim o sistema executará a ordem. No caso, ele vai buscar o primeiro garantidor do conjunto de garantidores e vai imprimir seu estado civil. Isso porque indexamos com a posição zero do vetor, que é a primeira posição em Lawtex.

Todos esses erros, desde os mais simples aos mais complexos nos levam ao que destacamos acima: A parte mais importante é a capacidade de fazer um raciocínio lógico, organizado e coerente.

A seguir vamos falar da topologia dos templates. Falamos em topologia pois o código em Lawtex é estruturado. Há um espaço certo para a inserção de cada tipo de código. Inicialmente isso pode parecer um formalismo excessivo. Por que razão eu sou obrigado a fazer declarações no início do código e só operá-las depois? Bom, há vários motivos para isso, mas o melhor deles é que quando terceiros forem ler seu código ficará mais fácil identificar quais informações são utilizadas e o que é feito com elas. Lembre-se que para efeitos de programação você mesmo no futuro é um terceiro. Depois de programar diversos templates dificilmente você vai se lembrar com exatidão de cada linha redigida. Então é preciso organização as informações para facilitar sua própria vida no futuro. Isso é verdade especialmente porque o código é um trabalho em constante evolução. Quase nunca você programa um template e nunca mais o edita. Novas teses podem surgir, erros podem aparecer, os usuários podem pedir novas funcionalidades. Nunca se sabe. Esteja, portanto, preparado:

1. Faça o código da maneira mais organizada que você conseguir.
2. Tente modularizar as lógicas (ex. criar um branch para cada argumento ou cláusula).
3. Não misture assuntos diferentes (não calcule num ponto do código um valor que você só vai utilizar adiante).
4. Dê nomes intuitivos para as variáveis (no lugar de chamar o número do processo de "<ndp>" use o alias "<numeroDoProcesso>").
5. Nunca abandone linhas de código antigas e desnecessárias no template.
6. Não mantenha uma lógica mais complexa e confusa só porque você já começou ela assim.

Adotando algumas dessas boas práticas de programação, um futuro de sucesso o aguarda engenheiro! Subjacente a todas essas recomendações está a seguinte: coragem! Não deixe o trabalho mal feito por preguiça de refazê-lo. É mais fácil corrigir e abandonar uma lógica no começo do que quando ela já está nas mãos do usuário. Isso é extremamente difícil. Os autores desse material já se deixaram (e ainda se deixam) seduzir pela preguiça diversas vezes e a conta sempre tem de ser paga meses depois.

Provavelmente esses conselhos não fizeram sentido algum para você se essa é a primeira vez que você leu o material, mas acredite, você vai lembrar delas ao longo dos próximos itens. Caso não lembre, volte e releia essa introdução depois que terminar de estudar todas as funcionalidades do sistema.

5.3 Topologia e Templates

Todo template em Lawtex tem 5 blocos importantes: **Metainfo**, **Head**, **Body**, **Foot** e **Extra**.

```
template[TEMP_TemplateAlias] {
    metainfo {
        language = "pt_BR"
    }
    head {
        title = "Contrato de prestação de serviços médicos"
        declarations {
            +<numeroDoContrato> : String {
                name = "Número do contrato"
                request = "Insira o número do contrato"
            },
            +<dataDoContrato> : Date {
                name = "Data"
                request = "Insira a data de celebração do contrato"
            }
        }
        operations {
            print bold("Número:") & " " & <numeroDoContrato>,
            print bold("Data:") & " " & <dataDoContrato>
        }
    }
    body {
        operations {
            print "Corpo do contrato..."
        }
    }
    foot {
        declarations {
            +<advogadoQueAssina> : String,
            +<oabDoAdvogado> : String,
            +<dataAssinatura> : Date
        }
        operations {
            print "Termos em que pede deferimento,\b\b",
            print <advogadoQueAssina> & "\b",
            print "OAB/SP nº " & <oabDoAdvogado>
        }
    }
    extra {
        operations {
            print "Template concluído."
        }
    }
}
```

Se atente na estruturação básica dos blocos do template. Na continuação segue uma descrição desses blocos.

Metainfo

O metainfo é o espaço em que você vai inserir as informações mais básicas e gerais do template, que valerão para todo o documento. Observe atentamente o código abaixo.

```
metainfo {
    language = "pt_BR"
    name = "Inicial de ação indenizatória contra empresa aérea"
    statement {
        audience = "Profissionais do direito especialistas em Direito do Consumidor"
        components = "O template usa os componentes globais da Looplex"
        functionalities = "O template avalia ilícitos cometidos pela empresa aérea."
        inputs = "Para preencher o template o usuário deve ter os dados do voo."
        overview = "O template redige uma ação indenizatória contra empresas aéreas"
        warnings {
            "Esse template não lida com acidentes aéreos",
            "Se houve  morte ou feridos esse template não se aplica"
        }
    }
    description = "Esse template recebe dados sobre falhas na prestação de serviços"
    style = "escritoriofamoso.lawsty"
    tags {
        "Consumidor", "Overbooking", "Indenização",
        "Aérea", "Voo", "Dano", "Moral", "Hipossuficiência"
    }
    declarations {
        +<parteAutora> : struct[ParteAutora] {
            name = "Dados da parte autora"
            request = "Insira os dados abaixo"
            fields {
                +[nome] : String {
                    name = "Nome"
                    request = "Qual o nome da parte autora?"
                },
                +[rg] : String {
                    name = "RG"
                    request = "Qual o RG da parte autora?"
                },
                +[cpf] : String {
                    name = "CPF"
                    request = "Qual o CPF da parte autora?"
                }
            }
        }
    }
}
```

Se existe alguma informação que você vai usar em diversas cláusulas ou diversos argumentos, declare essa informação no metainfo. Em geral a maior parte das informações será declarada no metainfo. Esse espaço não tem, como você poderá observar, parte operativa, pois as operações são feitas ao longo do texto. Esse é um espaço eminentemente declarativo e tudo que é declarado aqui é visível para o que está no resto do template.

Vamos passar ponto a ponto e entender o que cada trecho de código quer dizer. Em primeiro lugar abrimos o metainfo com chaves. Tudo que há dentro dele está delimitado por elas.

Na Linha 2 vemos um espaço para escolher a língua, o **language**. Esse espaço existe, pois o sistema permite tradução de documentos ou sua redação em outras línguas, então ele poderia ser preenchido, por exemplo, por "en_US".

Em seguida, na Linha 3, há um espaço para o nome do documento gerado, **name**. Esse nome é que aparecerá na lista de documentos disponíveis no sistema.

Em seguida temos o **statement** (Linhas 4 até 14), como ele tem diversos elementos ele também os delimita com chaves. Esses elementos serão textos mostrados ao usuário que está pensando em usar o template. Na interface ele verá isso como um resumo que o informa se esse é o documento que ele quer gerar ou não. Em primeiro lugar inserimos o público alvo, estabelecendo quais são os conhecimentos necessários para preencher o questionário. Em seguida identificamos os componentes, ou seja, trechos de código já programados por outras pessoas e só utilizados neste template. Em funcionalidades descrevemos quais lógicas jurídicas o sistema será capaz de fazer nesse caso e em inputs descrevemos quais informações são necessárias para preencher o questionário, orientando o usuário na coleta do material que ele usará para responder as questões. Em overview descrevemos brevemente o que é o template e em warnings fazemos ressalvas ou alertas sobre o uso do template. Encerrados os elementos que estão no statement, seguimos com os elementos gerais.

Na Linha 15, temos a descrição (**description**) se assemelha ao statement, mas tem uma destinação interna, de orientação de quem lê o código. Descreve brevemente o que faz e como funciona o código.

O **style** (Linha 16) indica o estilo de letra, tamanho de fonte, utilização ou não de parágrafos recursivos etc que serão utilizados no template. Normalmente cada escritório ou empresa possui estilo diferente para cada template produzido.

As **tags** (entre as Linhas 17 e 20) servem para orientar a busca no sistema. Quando alguém usar o mecanismo de busca com qualquer dessas palavras chave o sistema recomendará esse template. Ele também pode orientar a produção de dados agregados pelo sistema (ex. gerar um relatório da porcentagem de documentos gerados que tratam de Direito do Consumidor).

Nas declarações (**declarations** da Linha 21), como já adiantamos, definimos nossa "caixa de ferramentas", as informações que vamos usar nas lógicas e impressões. No exemplo declaramos um tipo complexo de informação que são os dados de um caso envolvendo empresa aérea. Logo em baixo declaramos que esse template terá uma variável desse tipo declarado. Nesse tipo declaramos em abstrato informações que todo caso envolvendo empresas aéreas deve ter.

No exemplo colocamos alguns campos demonstrativos. Primeiro perguntamos o nome da parte. Em seguida pedimos o RG. Ao final, pedimos o CPF da parte autora. É claro que em um template completo faríamos

mais perguntas. Perguntaríamos dados do caso, como por exemplo, se a parte autora perdeu o voo, se o voo mudou de portão, se ela estava atrasada, se havia fila etc. O objetivo é sempre afunilar as possibilidades e tentar inferir o caso por respostas objetivas do usuário.

Veremos a lógica de perguntas e operações com mais atenção em outros tópicos. Por enquanto, a lição que deve ser tomada é que esse é o espaço para declaração de informações que serão utilizadas ao longo de todos os argumentos.

Head

O **head** é um espaço de declaração de cabeçalhos. Números de processos, títulos, logotipos de escritórios, endereçamentos e outras informações desse tipo, quando se restringem apenas ao cabeçalho, devem ser declaradas e operadas no head. Além disso aqui definimos um título do documento, que aparecerá em destaque no topo da primeira página:

```
1  head {
2      title = "Instrumento particular de contrato de prestação de serviços médicos"
3      declarations {
4          +<numeroDoContrato> : String {
5              name = "Número do contrato"
6              request = "Insira o número do contrato"
7          },
8          +<dataDoContrato> : Date {
9              name = "Data"
10             request = "Insira a data de celebração do contrato"
11         }
12     }
13     operations {
14         print bold("Número:") & " " & <numeroDoContrato>,
15         print bold("Data:") & " " & <dataDoContrato>
16     }
17 }
```

Neste exemplo, além do título na Linha 2 e o bloco de declarações (Linhas 3 até 12), se observa a presença do bloco de operações (**operations**), onde as informações que foram declaradas são usadas nas lógicas e impressões. No exemplo estamos imprimindo o número e a data do contrato. Neste exemplo, duas variáveis foram declaradas, uma do tipo texto (String) e outra do tipo data (ver Seção 5.4).

Body

O **body** tem a exata mesma estrutura que o head, como é possível ver no snippet do template. Ele é o espaço dedicado à impressão do corpo do documento, o texto em si, com argumentos, cláusulas, tópicos, parágrafos etc. Também é nele que fazemos avaliações e lógicas necessárias para a realização dessa impressão. Exemplo:

```
body {
    declarations {
        +<devedor> : *Devedor,
        +<dataAtraso> : Date {
            name = "Início do atraso"
        },
        +<valorDevido> : Currency {
            name = "Valor devido"
        }
    }
    operations {
        if (<devedor.genero> == "Masculino") {
            print "Prezado "
        } else {
            print "Prezada "
        },
        print <devedor.nome> & ",\b\b",
        print "Verificamos em nosso sistema que você tem parcelas em atraso
               desde " & <dataAtraso> & ". ",
        print "Atualmente o valor total devido, acrescido de juros e multa,
               é de R$ " & <valorDevido> & ". Entre em contato conosco o mais
               rápido possível com a nossa equipe para apresentar os seus
               comprovantes de pagamento ou quitar as parcelas devidas."
    }
}
```

No exemplo fizemos o corpo simples de uma notificação de parcelas em atraso. Declaramos o que precisávamos: um devedor, com suas características, uma data de início do atraso e um valor total devido. Em seguida imprimimos um texto, concatenando com as informações necessárias (o que é feito por meio do símbolo "&" entre os textos).

Foot

O **foot**, mais uma vez, tem a mesma estrutura que o body e o head.

```
foot {
    declarations {
        +<advogado> : *Advogado
    }
    operations {
        print "Termos em que pede deferimento,\b\b",
        print sign(<advogado.nome>) & "\b",
        print "OAB/" & <advogado.oab.uf> & " nº " & <advogado.oab.numero>
    }
}
```

Todas as considerações feitas acima valem para ele. Em geral ele é o encerramento do documento:

Extra

O **extra** é um espaço exclusivamente operativo que se destina a realizar operações depois que o documento é utilizado. Sua principal função é ser o espaço de execução das funções de *smart documents*. Na data de redação deste material, ele ainda era pouco utilizado, mas é o espaço privilegiado para dar comandos que mandem e-mails, registrem andamento de processos no sistema de gestão do escritório etc. Não abordaremos nessa oportunidade.

5.4 Classes de operandos

Conforme observado nos exemplos de código anteriores, dentro das declarações constam unidades de armazenamento do valores. Tais unidades são os operandos.

Operandos são *containers* de informações, repositórios onde se acessa dados variáveis, sejam obtidos a partir de respostas do usuário, sejam obtidas por meio de cálculos e operações dentro do próprio algoritmo. Existem informações atômicas que são armazenadas em **variáveis**, assim como existem informações que não tem nenhum acesso externo ao sistema, ou seja, são **constantes** dentro do código. Existem operandos que representam coleções de dados, sejam elas homogêneas, como no caso de **vetores** ou variáveis do tipo lista, ou sejam elas heterogêneas, como os casos de **objetos** do tipo estrutura.

Variáveis e objetos

Utilizamos a variável para armazenar informações inseridas pelo usuário. Declaramos uma variável quando pretendemos imprimi-la ou utilizá-la em alguma lógica ou avaliação ao longo do código. Sempre que declaramos a variável precisamos informar ao sistema seu tipo, além de suas propriedades, já descritas acima. Nos itens acima vimos diversos exemplos de variáveis com suas propriedades. Aqui vamos falar um pouco mais de seus tipos.

Uma variável pode ser: (1) String, (2) Text, (3) Integer, (4) Real, (5) Currency, (6) List, (7) Boolean, (8) Date e (9) Time. Além desses tipos, chamados primitivos, a variável pode ser do tipo (10) Struct. Vamos passar por cada tipo com mais detalhes em tópicos seguintes, mas por enquanto saiba que (1) e (2) são campos livres para o usuário digitar o que quiser, (3), (4) e (5) são números, (6) é uma lista de Strings, como já descrevemos, (7) é qualquer pergunta cuja resposta é "sim" ou "não", (8) é uma data e (9) um horário. Como também já adiantamos (10) é um objeto, ou seja, um tipo customizado pelo usuário. Pessoa, endereço, processo judicial são exemplos de tipos, com várias informações a eles vinculadas. Uma variável do tipo pessoa gerará um *card* que pergunta CPF, nome, endereço, telefone, e-mail etc.

Quando uma variável possui o tipo Struct, suas propriedades serão aquelas definidas na Struct para todas as variáveis daquele tipo. Em particular, uma variável que possui um tipo Struct é chamada de Objeto. Note também que é frequente termos mais de uma variável com o mesmo tipo Struct. Podemos, por exemplo, ter uma variável chamada <locador> e outra chamada <locatario> ambas do tipo pessoa. Por fim, é preciso atenção na redação do código para usarmos a sintaxe correta.

As variáveis são delimitadas pelos sinais '<' e '>'. Muito parecidos com as variáveis são os campos. Como já sugerimos os campos são, de certa forma, as variáveis associadas a um objeto. As **variáveis** e os **objetos** são informações avulsas e os **campos**, delimitados pelos colchetes '[' e ']' são informações declaradas dentro de uma Struct, como veremos adiante. Da mesma forma, é preciso atenção para não confundirmos os delimitadores de variáveis com os delimitadores de **vetores**, que são cercados pela barra '|'. Seguem dois exemplos de variáveis de diferentes tipos:

```
+<locador> : *Sujeito,
+<temGarantia> : Boolean
```

Exercício 32. A seguir, vamos fazer exercícios para fixar o conceito de variável.

a) Faça uma variável de provas a serem apresentadas pelo autor em uma contestação bancária.

b) Faça uma variável que questiona o usuário acerca da realização de audiência de conciliação ou não.

Vector

Já sugerimos o que é o vetor ao descrevermos suas propriedades. O vetor é melhor entendido como um conjunto de variáveis de mesmo tipo. Quando usamos uma variável para pedir ao usuário o autor de uma ação ele só poderá inserir os dados do autor uma vez. Quando usamos um vetor permitimos que o usuário repita o procedimento de inserção dos dados de um autor quantas vezes julgar necessário, para o número de autores que desejar.

É frequente o uso de vetores em documentos jurídicos. Podemos ter um número indefinido de partes, de vendedores, de autores, de réus, de intervenientes anuentes etc. Daí a necessidade de representá-los como vetor. Como já indicamos, o vetor deve ser um conjunto de elementos do mesmo tipo, logo devemos definir o tipo do vetor. Vejamos os exemplos a seguir:

```
+|autores| : Vector[*Autor],
&|telefones| : Vector[String]
```

Lembre-se que é possível criar vetores tanto de maneira avulsa quanto como um campo de um objeto. Podemos declarar um vetor de autores e cada um desses autores, dentre os campos que os definem, ter um vetor de telefones. Isso muitas vezes provoca confusão, mas é um aspecto muito utilizado do código. Assim como cada um dos autores de um conjunto de autor tem o próprio nome e o próprio CPF ou CNPJ, cada um deles pode ter seu próprio conjunto de endereços, de telefones, de e-mails etc.

Como já adiantamos na introdução da parte sobre sintaxe, o uso de vetores deve ser indexado ou iterado. Se eu desejo, por exemplo, imprimir o nome de cada um dos autores eu tenho que proceder da seguinte forma:

```
foreach(<autor> IN |autores|) where (separator = "%f1, %s2, %p2, bem como %l2.") {
    print {
        [version = <autor.nome> & ", portador do documento (RG) nº " & <autor.rg>],
        [version = <autor.nome> & ", RG nº " & <autor.rg>]
    }
}
```

Basicamente, essas linhas de código falam para o sistema tomar a sequência de autores e, para cada um deles, imprimir seus respectivos nomes, separados por vírgulas entre o primeiro e o penúltimo elemento, por um "e" entre o penúltimo e o último, finalizando com um ponto final. Explicando o que significa o argumento **separator** dentro da cláusula where, lê-se %f como *first*, %s como *second*, %p como *penult* e %l como *last*. O número ao lado dessas letras diz qual versão do print deve-se imprimir. Assim sendo, a primeira vez irá imprimir "..., portador do documento (RG) nº ...", enquanto da segunda vez em diante imprimirá somente "..., RG nº ...".

No caso de vetor ainda se admite acesso a um elemento específico (acesso indexado). É possível fazer o seguinte:

```
print |autor{0}.nome|
```

Ou seja, pedir que o computador imprima o nome do primeiro autor do conjunto de autores, ou seja, a posição zero do vetor. Note que as posições dos vetores sempre se iniciam em 0 (zero).

Faremos mais três comentários adicionais sobre o vetor. Em primeiro lugar, podemos saber quantas posições tem o vetor usando o comando **size**. Esse recurso é muito útil quando queremos saber, por exemplo, se há só uma parte ré ou mais de uma delas. Para tanto basta fazermos "|reus|.size()". Podemos usar esse valor para fazer avaliações ou até imprimi-lo.

Outro recurso interessante é que podemos pedir para que o usuário escolha um dos objetos complexos que ele inseriu anteriormente. Ao invés do usuário se deparar com todos os dados de todos os objetos ele vai se deparar com uma lista resumida, em que pode escolher qual deles quer. Veremos mais detalhes dessa funcionalidade, chamada **select** quando discutirmos estruturas. Por enquanto, imagine que você tem um vetor de advogados num template e quer que o usuário escolha quais deles receberão as intimações:

```
|advogados|.select(|advogadosIntimados|,"Quais advogados foram intimados?",0,1)
```

Note que dentro dos parênteses, além do vetor que recebe os elementos selecionados dos primeiros e da pergunta feita, há mais outros dois parâmetros ao final, separados por virgulas. Trata-se dos limites, o lower (no mínimo) e o upper (no máximo). O primeiro indica quantos elementos no mínimo o usuário deve escolher. O segundo indica quantos elementos no máximo o usuário deve escolher.

Por fim, é possível filtrar o vetor, acessando um subconjunto dele que satisfaz determinadas características. É o caso quando queremos acessar apenas as partes casadas, para depois fazer uma cláusula de representações:

```
|partesCasadas| = |partes|.filter (
    [this.tipoDeSujeito] == "Pessoa natural" AND [this.estadoCivil] == "Casado"
)
```

Veja que o vetor de pessoas casadas recebe uma versão filtrada do vetor de partes. O critério do filtro é pegar qualquer elemento (indicado genericamente pelo **this**) que seja pessoa natural e que seja casado.

Exercício 33. Coloque em prática o que você aprendeu até aqui. Faça a declaração e a operação de um vetor de advogados que assinam uma petição.

Constant

A constante se assemelha à variável em vários aspectos. Ela também tem um tipo, também pode ser operada e impressa como a variável, mas é utilizada em geral para informações que não são preenchidas pelo usuário e se mantém constantes ao longo do código. Ela é cercada pela cerquilha '#', conforme mostraremos a seguir. A constante também tem como propriedades: tipo, nome e ajuda. Vejamos a sintaxe da constante:

```
-#tipoDeContrato# : String {
    nome = "Tipo de contrato"
    value = "Contrato de Locação"
}
```

Observe que a definição da constante apresenta uma propriedade **value** recebendo um valor (pelo símbolo de igual), dentro de chaves colocadas imediatamente após o tipo String. Esse bloco delimitado por chaves é a área de definição de propriedades do operando. Independentemente do operando, em Lawtex eles têm suas propriedades particulares.

Uma diferença importante entre a constante e os demais operandos, é que no lugar do default ela tem uma outra propriedade chamada value. Na prática seu funcionamento é o mesmo. Os dois são definições prévias da informação pelo engenheiro ou pela engenheira. A diferença é que o default é um padrão, uma predefinição do valor para agilizar o preenchimento pelo usuário que usa frequentemente o padrão. O value, embora também seja uma informação predefinida, não se destina a ser substituído pelo usuário. Ele é efetivamente o valor que o engenheiro quer que essa constante tenha. Um uso possível da constante ocorre no uso de componentes globais já programados.

Suponha que você esteja programando um contrato e que esse contrato usará a funcionalidade de cláusulas de garantia. Os tipos de garantias acessíveis ao usuário dependem do tipo de contrato. Alguns contratos, como o de locação, não admitem certos tipos de garantia (art. 37 da Lei de Locações). Dessa forma, é interessante que todo contrato tenha uma constante que armazene o tipo de contrato. Assim, o componente de garantias, quando utilizado, saberá o tipo do contrato e ajustará automaticamente as garantias possíveis. Isso também pode ser feito com uma variável oculta, mas o código fica mais claro se isso for feito por constante.

5.5 Propriedades dos Operandos

Nesse ponto do texto nosso foco será quais são essas propriedades, sua utilidade e como expressá-las. Ao final deste item descreveremos onde e como essas propriedades aparecem em cada um dos tipos de operandos.

Mandatory

Iniciaremos com a propriedade mandatory. Essa propriedade tem uma "responsabilidade" dupla no código Lawtex. Ela governa: (a) a obrigatoriedade de preenchimento de um campo ou informação pela usuário e (b) a visibilidade do campo ou informação pelo usuário. Essa propriedade comporta três opções:

(1) visível e obrigatório, representada pela sintaxe '+';

(2) visível, mas opcional, representada pela sintaxe '&' e, por fim;

(3) invisível, representada pela sintaxe '−'.

Vejamos exemplos das três opções. Suponha que queiramos perguntar ao usuário, por meio de uma variável, o número do processo para fazer o endereçamento de uma contestação. Essa informação é indispensável nesse contexto, de forma que os jovens engenheiros jurídicos devem obrigar o usuário a preenchê-la. Logo, essa variável seria declarada da seguinte forma:

```
1 +<numeroDoProcesso> : String
```

Há duas observações necessárias aqui. Em primeiro lugar, as propriedades são definidas apenas na declaração. Na operação, elas serão dadas, ou seja, o sistema já saberá que aquele operando tem as propriedades declaradas, de forma que não é necessário repeti-las. É possível, contudo, alterá-las por meio de operações, como demonstraremos nos tópicos seguintes. A segunda é que a propriedade mandatory fica sempre a esquerda do alias do operando declarado (o '+' do exemplo acima), sendo o lado direito reservado ao tipo do operando, o que também discutiremos adiante.

Seguiremos com outro exemplo. Suponha, agora, que vamos pedir o endereço de alguém. Alguns endereços têm complemento (ex. conjunto A, sala 2, 5º andar, apartamento 12 etc.), mas outros não. Assim, se exigirmos que o usuário insira a informação vamos gerar um problema para quem mora em uma casa fora de um condomínio! A informação deve, portanto, ser opcional:

```
1 &<complemento> : String
```

Nesse caso o usuário poderá ou não inserir o dado, ficando a seu critério salvar sem resposta o campo opcional.

Por fim, suponha que queiramos calcular o valor total de um contrato. Esse contrato tem parcelas e todas parcelas têm o mesmo valor:

```
+<numeroDeParcelas> : Integer {
    name = "Número de parcelas"
},
+<valorDaParcela> : Currency {
    name = "Valor da parcela (em R$)"
}
```

É fácil ver que seria redundante perguntar ao usuário qual é o valor total do contrato. Se sabemos que as parcelas têm sempre o mesmo valor e se conhecemos o valor e o número de parcelas, basta multiplicar as variáveis para obter o total. Logo, ainda que precisemos da informação "valor total", nós calcularemos nós mesmos, sem que seja necessário perguntar ao usuário. Logo, essa informação será oculta e permanecerá oculta até que o próprio sistema faça a conta e a responda, sem nenhuma interferência do usuário:

```
+<numeroDeParcelas> : Integer,
+<valorDaParcela> : Currency,
-<valorTotalDoContrato> : Currency
```

Como veremos em tópicos futuros, podemos realizar operações matemáticas e inserir seus resultados dentro de variáveis, ocultas ou não, por meio de operações como a seguinte:

```
<valorTotalDoContrato> = <numeroDeParcelas> * <valorDaParcela>
```

Não se preocupe com as operações. Dedicaremos uma parte desse texto apenas para elas. Por enquanto, queremos apenas que você entenda qual a situação que demanda um operando visível e obrigatório, visível e opcional e invisível.

Exercício 34. Escreva, a seguir, como você declararia as seguintes variáveis, usando apenas a propriedade até agora ensinada:

a) Valor da apólice de um seguro, dentro de um contrato de seguro.

b) Telefone do locatário, dentro de um contrato de locação.

c) Valor atualizado de uma dívida cobrada.

Name

A propriedade name também se aplica a todos os operandos. Ela se destina, sobretudo, a orientar o usuário sobre qual é a informação ou pergunta presente em um campo ou variável a ser preenchida. Assim, sempre que o sistema for se referir a uma pergunta específica, ele vai usar o nome da pergunta, e não a pergunta na íntegra. Isso ficará mais claro com exemplos.

Na interface há um sumário com uma lista das informações que serão preenchidas. Imagine um template que pede ao usuário o valor da multa por atraso no pagamento. A pergunta desse campo é "Insira, em

porcentagem, o valor da multa pelo atraso no pagamento". Como vimos, a pergunta tem que ser o mais clara o possível para prevenir erros do usuário. Ocorre que, justamente por isso, ela é longa demais para que seja referência no sumário. Logo, no sumário, a variável que receberá essa informação será identificada pelo nome "Multa por atraso". Assim, o usuário identifica do que trata aquela variável e visualiza a pergunta completa após selecioná-la. O nome da variável também é utilizado na revisão.

O *card* de revisão, como já adiantamos, é um resumo de todas as informações que foram preenchidas. Assim como no sumário, não faz sentido colocar a pergunta completa numa lista sucinta das informações que já foram preenchidas. Logo, na tabela do *card* de revisão o usuário verá, na coluna da esquerda, o nome dos campos e variáveis que preencheu e, no lado direito, a informação inserida. Seguiremos agora com a sintaxe necessária para definir a propriedade name.

Note que no exemplo a seguir inclui o tipo da variável, a propriedade estudada no item anterior e a próxima propriedade que veremos:

```
1 +<numeroDeParcelas> : Integer {
2     name = "Número de parcelas"
3     request = "Insira o número de parcelas do financiamento"
4 }
```

Assim, ao executar esse código, o sistema gerará uma variável para ser preenchida pelo usuário. No sumário e no *card* de revisão ela será identificada pelo nome "Número de parcelas". Ao clicar em "Número de parcelas" o usuário irá, em seguida, para a tela de preenchimento da variável, que pede que o usuário "Insira o número de parcelas do financiamento" e oferece o espaço para que ele o faça. Como veremos adiante, ao declararmos a variável com essa propriedade ela se manterá com esse nome independentemente de como usarmos a informação armazenada pela variável ao longo do documento. Lembre-se, contudo, como também já adiantamos, que é possível alterar essa propriedade ao longo da parte operativa. Descreveremos a sintaxe dessa modificação nos tópicos a seguir. Por ora seguiremos discutindo em mais detalhes o request a seguir.

Um último apontamento necessário é que o name é diferente do alias. Para compreender a diferença basta fazer uma analogia com seus documentos no computador: name é o título deles quando você os abre com um editor de texto, ao passo que alias é a extensão do arquivo, ou seja, seu nome do ponto de vista do sistema. O nome é sempre utilizado na interface e com o usuário. Já o alias é o nome que aparecerá no sistema.

Exercício 35. Declare, a seguir, definindo name e request nas seguintes variáveis:

a) Número do processo CNJ, com pontos e traços.

b) Valor original da dívida, sem multa, juros ou correção monetária.

Request

Como já indicamos, o request é o texto mostrado ao usuário logo acima do campo em que ele insere uma informação. Seu papel é descrever da melhor forma possível para o usuário o que queremos dele, ou seja, que informação ele deve inserir. Ele deve ser curto e de fácil entendimento. Retomando o que descrevemos

nas reflexões teóricas sobre engenharia jurídica, a interpretação que o usuário faz do request é diferente da interpretação que os juristas costumam fazer da lei. Na interpretação do request o usuário deve, de fato, entender a intenção do engenheiro ou da engenheira que programou o template. A melhor forma de garantir que essa seja a interpretação do usuário é pedir uma informação fática e simples.

Devemos justamente evitar perguntas jurídicas e abstratas, pois o exercício da engenharia é deixar a abstração e a complexidade do Direito para o sistema, restando para o usuário atividades simples. No lugar de perguntarmos, por exemplo, "a multa é abusiva?" podemos perguntar diretamente "qual é o valor da multa" e contrastarmos esse valor com o total da obrigação ou com algum parâmetro que nós temos para que o sistema determine sozinho se a multa é ou não abusiva. O usuário não deve nem saber que o sistema está fazendo essa avaliação quando ele insere a informação objetiva pedida, qual seja, o valor da multa. Um exemplo da sintaxe do request foi dado no item anterior.

Exercício 36. Declare, abaixo, uma variável indicando o name e request:

a) A indicação do tipo de acordo (extrajudicial ou judicial) a ser celebrado.

Key

A ideia do key é informar o sistema que aquela informação pode ser usada como referência para acionar o cadastro. O que isso quer dizer? Como veremos, podemos pedir "informações avulsas" com variáveis ou podemos criar objetos, ou seja, um agregado de informações vinculadas entre si. O que é uma pessoa? Um número? Um texto? Uma lista? Nenhum deles. Uma pessoa, do ponto de vista do sistema, é uma agregação de informações. Uma pessoa tem nome, endereço, CPF ou CNPJ dependendo se for pessoa física ou jurídica e assim por diante. Logo, essa pessoa é um agregado de informações. Seguimos agora com a sintaxe:

```
+[rg] : String {
    name = "Número de RG"
    request = "Insira o número de RG com pontos e traços"
    key = true
},
+[estadoCivil] : List ("Solteiro", "Casado", "Convivente estável", "Divorciado",
                       "Viúvo") {
    name = "Estado civil"
    request = "Indique o estado civil"
    key = false
    atomic = true
}
```

A propriedade key é utilizada para recuperar informações de objetos a partir de campos. Ela indica que uma (ou mais) informações desse objeto deve ativar o cadastro. Retomando o exemplo de pessoa, cada pessoa só tem um CPF, que será apenas dela. Assim, posso informar ao sistema que ele é key. Com essa informação quando o usuário inserir o CPF da pessoa o sistema busca no banco de cadastro alguma pessoa com esse CPF. Se ele encontrar ele saberá que se trata da mesma pessoa e trará todos os demais dados da pessoa já inserida, poupando esforço do usuário no preenchimento de informações.

Note que o uso dessa propriedade exige cautela. Devemos utilizá-la em uma ou mais informações de um objeto e devemos usar em uma informação exclusiva. Mais de uma pessoa pode morar na mesma rua, logo, rua não é uma informação key. Já o CPF, o CNPJ, o RG, o número do passaporte, o Social Security Number, entre outros são key de uma categoria específica de objetos, os sujeitos. O garantidor, o contratante, o investidor, o credor, o locador etc. Todos esses objetos são sujeitos e, como sujeitos, têm um número que os identifica.

Não se preocupe com propriedades ainda não vistas. Inserimos dois campos diferentes propositalmente. Isso porque queremos deixar claro que essa propriedade é usada no contexto de objetos ou, como veremos adiante, de Structs e que nem toda informação associada a esse objeto é key. Observe no exemplo que RG tem key igual a true e estado civil igual a false. Isso porque o estado civil não identifica uma única pessoa no cadastro, mas o RG sim.

Exercício 37. Responda as questões abaixo:

a) CEP é key de um endereço?

b) Qual campo deve ser key de um processo judicial?

Atomic

Para explicar essa propriedade vamos retomar o exemplo do estado civil. Isso porque atomic é uma propriedade exclusivamente aplicável às listas. Dedicaremos um item apenas às variáveis e campos do tipo lista, mas faremos uma pequena introdução aqui para facilitar a compreensão da propriedade atomic. Quando definimos uma variável como tipo lista também definimos diversos textos como suas opções. Assim, limitamos a ação do usuário à escolha de um desses textos. Há dois tipos de listas: as atômicas e as não-atômicas.

A propriedade aqui discutida irá definir o tipo da lista. A lista atômica é aquela em que a escolha de uma das alternativas exclui a escolha das demais. Quando pedimos ao usuário gênero para fazermos as concordâncias ele não pode escolher masculino e feminino ao mesmo tempo. Configuramos a lista de gêneros para ser atômica para que ele escolha uma só opção. Por outro lado, quando perguntamos ao usuário o que foi pedido numa petição inicial e oferecemos uma lista dos pedidos possíveis o usuário poderá escolher, simultaneamente, quantos quiser. Vejamos exemplos da sintaxe da propriedade:

```
+<statusDivida> : List ("Em atraso", "Em dia", "Quitada") {
    name = "Status da dívida"
    request = "Selecione o status do pagamento das parcelas da dívida''
    atomic = true
},
+<pedidos> : List ("Remoção do cadastro de inadimplentes", "Anulação do contrato",
                   "Outro") {
    name = "Pedidos"
    request = "Pedidos da parte autora"
    atomic = false
}
```

Veja que o pagamento da dívida não pode estar atrasado e em dia ao mesmo tempo. Da mesma forma, uma dívida ou está integralmente quitada ou o pagamento das parcelas até o momento está em dia. Assim, não podemos permitir que o usuário escolha mais de uma opção, então a propriedade atomic é igual a true, ou seja, ele deve informar se o pagamento está em atraso, se ele está em dia, mas ainda faltam pagamentos a ser realizados ou se toda a dívida já foi quitada. Já nos pedidos o mesmo não ocorre, a lista não tem de ser atômica, pois é possível que na inicial a parte autora peça uma das três opções, duas delas ou todas. Lembre-se que a parte pode pedir anulação do contrato e, subsidiariamente, a revisão do valor das parcelas.

Outro ponto interessante é que em diversas iniciais há pedidos contraditórios. Logo, como veremos adiante, devemos permitir que usuário consiga informar ao sistema as contradições e fraquezas do argumento da outra parte. O usuário não precisa entender essas fraquezas. O próprio sistema estará programado para reconhece-las a partir das informações do usuário.

Exercício 38. Declare abaixo os seguintes campos:

a) Campo em que o usuário informa o tipo societário de uma empresa.

b) Campo em que o usuário informa, dentre extração, agropecuária, indústria, e serviços os setores em que a empresa atua.

Upper

Para explicar a propriedade upper, assim como fizemos para outras propriedades, precisamos apresentar o contexto em que ela surge. O principal uso do upper é com vetores. Vetores, como já sugerimos, são conjuntos. Eles podem ser conjuntos de qualquer coisa. Podemos ter um vetor de textos, em que o usuário digita cada atividade do objeto social de uma empresa e até um vetor de objetos, em que o usuário insere os endereços de suas filiais.

Essa propriedade coloca um limite máximo de elementos que usuário pode inserir no conjunto. Isso porque, em geral, quando declaramos um vetor, o usuário pode inserir quantos elementos ele quiser. Um contrato pode, a princípio, ter quantas partes o usuário desejar. Ocorre que em alguns contextos podemos querer que usuário possa inserir um número de elementos que quiser, respeitado determinado limite. Imagine que queremos que o usuário insira os sócios de uma sociedade limitada, mas queremos impedir que ele insira mais de dez sócios. A sintaxe seria a seguinte:

```
+|socios| : Vector[*Socio] {
    name = "Sócios"
    request = "Insira os dados de cada sócio"
    upper = 10
}
```

Não se incomode com os elementos que você ainda não conhece dessa sintaxe. Eles são os elementos de um vetor de objetos. No caso, o usuário poderá inserir diversos sócios, sendo os campos definidos a parte numa struct de sócio. Veremos isso com mais detalhes adiante. Aqui o que importa é que quando o usuário

visualizar esse vetor ele não verá nenhum elemento no vetor. Apenas um botão com o ícone '+'. Selecionando esse ícone ele acrescentará um elemento. Ele poderá fazer isso até o máximo de dez elementos.

A propriedade também pode ser utilizada com listas não-atômicas para limitar o número de opções da lista que o usuário pode escolher simultaneamente. Assim, a lista pode ter, por exemplo, cinco opções, mas o usuário pode ser constrangido a escolher no máximo três delas.

Exercício 39. A seguir, declare o número máximo de pessoas físicas que podem constituir uma empresa individual de responsabilidade limitada (Eireli).

Lower

O lower tem a mesma função do upper mas para impor um limite mínimo de elementos do vetor. Esse limite costuma ser mais utilizado que o upper. É frequente obrigarmos o usuário a colocar pelo menos uma parte autora, no mínimo duas testemunhas, pelo menos uma garantia etc. Ele também pode impor que usuário escolha, por exemplo, pelo menos duas opções de uma lista de cinco opções. Vamos usar o exemplo das testemunhas:

```
+|testemunhas| : Vector[*Testemunha] {
    name = "Testemunhas"
    request = "Insira os dados das testemunhas que assinarão o contrato"
    lower = 2
}
```

Exercício 40. Vamos lá! Declare o número mínimo de sócios para a constituição de uma sociedade empresária de responsabilidade limitada.

Default

O default é uma propriedade muito utilizada e importante. Fundamentalmente o default se destina a predefinir uma resposta padrão para tornar o preenchimento mais rápido para a maior parte dos usuários. Um exemplo é quando pedimos o endereço da parte em algum contrato ou petição. Como o sistema é brasileiro e se destina primordialmente a automatizar documentos conforme o Direito Brasileiro, é de se esperar que a maioria das partes inseridas no sistema sejam brasileiras, residentes e domiciliadas no território nacional. Assim, como a grande maioria dos usuários irá escolher "Brasil" como país no momento de preencher o endereço, é razoável deixar o Brasil como resposta padrão predefinida, poupando a maior parte dos usuários de preencher esse campo, sem impedir que os poucos casos de pessoas sediadas ou domiciliadas em outros países sejam cobertos.

Para além disso, como discutiremos com mais detalhes no item de objetos ou estruturas, o default tem um papel importante para evitar erros de renderização. Esse ponto será crucial quando tratarmos de polimorfismo. Para os fins desse tópico é suficiente ter em mente que muitas vezes não podemos deixar um

campo vazio. Como vimos, as estruturas ou objetos têm campos. Esses campos podem surgir dinamicamente. O sistema escolhe pedir ao usuário o CPF ou o CNPJ de uma parte dependendo de uma resposta anterior do usuário no sentido de que essa parte é pessoa física ou não. Como já dissemos, o sistema tem uma ordem de execução das ordens que damos a ele. Se no momento que ordenamos, por exemplo, que o sistema imprima o CPF da pessoa e o sistema ainda não sabe se a pessoa é física ou não, ele vai acusar um erro. É preciso garantir ao sistema que a informação utilizada efetivamente existe. Um dos instrumentos para garantir a existência de uma informação no início da execução, ou seja, antes do usuário responder, é colocar um default na pergunta condicionante. Isso porque enquanto a condicionante for vazia o sistema não sabe avaliar qual das opções ele deve escolher. Vejamos um exemplo:

```
1 +[tipoDeSujeito] : List ("Pessoa natural", "Pessoa jurídica",
2                          "Ente não personificado") {
3     name = "Tipo de sujeito"
4     request = "A parte autora é..."
5     key = false
6     atomic = true
7     default = "Pessoa jurídica"
8 },
9 if ([tipoDeSujeito] == "Pessoa natural") {
10     +[numeroReceita] : String {
11         name = "CPF"
12         request = "Insira o número de CPF da parte autora"
13         key = true
14     }
15 } else {
16     +[numeroReceita] : String {
17         name = "CNPJ"
18         request = "Insira o número de CNPJ da parte autora"
19         key = true
20     }
21 }
```

Como podemos ver o default (Linha 7) garante que pelo menos um dos tipos de número perante a Receita Federal existe. Do contrário, no início da execução, antes do usuário preencher o documento, nenhum dos dois campos existiria e a ordem de impressão resultaria num erro. Logo que fosse iniciado, o sistema acusaria um erro.

Além disso, o default também facilita, pois o usuário não tem que mudar nada se a parte for pessoa jurídica. Por fim, adiantando um pouco do que será discutido sobre polimorfismo, como existem dois campos com o mesmo nome, eles devem estar em condições excludentes. Não é possível existir dois campos com o mesmo nome. Se você declarar, você deve garantir que quando um existir o outro não existirá.

Exercício 41. Descreva a sintaxe, utilizando a propriedade default para que o sistema possa imprimir o regime de bens, caso a pessoa física seja "casada".

Help

O help complementa o request. Lembre-se que quando tratamos do request dissemos que ele deve ser curto, pois de fato as perguntas devem ser objetivas e, para que a tela não fique poluída, há pouco espaço para elas. Não é desejável que tenham mais de 150 caracteres. Se faltar uma descrição mais específica ou uma explicação melhor da dinâmica de resposta do campo, é possível acrescentar uma ajuda. A ajuda pode ter um texto maior e ser bem detalhada. Ela é representada por uma pequena interrogação e pode ser aplicável a um *card* inteiro ou a um campo de uma estrutura. Assim como as outras propriedades, o help é definido nas declarações, mas pode ser alterado com operações ou com polimorfismos para se adequar ao contexto em que o usuário se insere. Vamos usar um exemplo muito frequente de contencioso:

```
+[temProcuracao] : Boolean {
    name = "Procuração ad judicia "
    request = "A parte autora juntou procuração?"
    help = "Verifique, nos autos, se há procuração da parte autora"
},
if ([temProcuracao] == true) {
    +[procuracaoEhAdequada] : Boolean {
        name = "Adequação da procuração"
        request = "A procuração é adequada? Veja o help para mais detalhes"
        help = "Nesse campo você deve responder 'não' se a procuração não é adequada.
                Considere adequada a procuração se a) ela foi assinada pela parte
                autora e b) se o advogado que assinou a peça é o que recebeu os
                poderes ou se houve substabelecimento daquele para o que assina"
    }
}
```

Veja que nesse exemplo o sistema só pergunta se a procuração é adequada se houver procuração e que com a pergunta há uma explicação de como identificar se a procuração é adequada. Vamos treinar o seu conhecimento.

Exercício 42. Escreva, a seguir, como você declararia a propriedade help para auxiliar o usuário na escolha do Estado da vara trabalhista competente para ajuizar determinada ação.

Máscaras

As máscaras são restrições, limites impostos ao usuário no exercício da liberdade de preencher algum campo aberto. É comum utilizá-las quando a informação que você deseja tem um formato específico. Um exemplo claro é o número CNJ para processos. Embora o usuário possa digitar o número, sabemos de antemão o número de dígitos e quantos pontos e traços deve ter o número que ele vai digitar. O mesmo vale para CPF e CNPJ. Assim, diminuímos, por exemplo, a chance de o usuário esquecer um dos caracteres e aumentamos a padronização, dado que o sistema mesmo vai inserir os caracteres fixos necessários. Vejamos um exemplo:

```
1 +[numeroDoProcesso] : String
2          where ("\d\d\d\d\d\d\d-\d\d.\d\d\d\d.\d.\d\d.\d\d\d\d") {
3     name = "Número do processo"
4     request = "Insira o número do processo"
5     key = false
6 }
```

A indicação de máscara vem após a diretiva *where* da Linha 2. Com essa sintaxe o sistema só permitirá que o usuário insira dígitos (representados pelo '\d'). Além disso ele colocará um traço após 7 dígitos, um ponto após os próximos dois dígitos e assim por diante. As máscaras só se aplicam às Strings, então mesmo que chamemos o identificador do CNJ de "número", do ponto de vista ele é, na verdade, um texto.

Os símbolos de máscara são:

\d Representa somente os dígitos: [0-9]

\a Representa somente as letras minúsculas: [a-z]

\A Representa somente as letras maiúsculas: [A-Z]

\@ Representa os alfa-numéricos, isto é, dígitos e letras maiúsculas e minúsculas: [0-9a-zA-Z]

_ Representa espaço em branco, tabulação ou quebra de linha: [\s]

\. Representa o universo de todos caracteres: [·]

\c Representa qualquer caractere, exceto letras maiúsculas: [^A-Z]

\C Representa qualquer caractere, exceto letras minúsculas: [^a-z]

\r Quando precede outro símbolo ou caractere, significa que a ocorrência do mesmo é recursiva, podendo haver nenhuma ou mais ocorrências.

\o Quando precede outro símbolo ou caractere, significa que a ocorrência do mesmo é opcional.

Exercício 43. Agora tente você! Faça a máscara para o número de CPF de uma pessoa física.

5.6 Declarações e Tipos

Todo operando está associado a um tipo específico, que restringe o que o usuário deve responder. Por exemplo, se um operando for numérico, o mesmo não pode admitir caracteres alfa numéricos. Ou ainda, se o operando for do tipo data, o usuário deve ter em mãos uma componente específica para seleção de datas. Deste modo, explicaremos cada um dos tipos existentes em Lawtex.

String

Como já adiantamos **String** é um tipo primitivo. Campos, variáveis e vetores desse tipo são textos livres, que o usuário pode preencher como quiser. Há algumas funcionalidades voltadas para textos, como veremos adiante. Podemos colocar textos em letra maiúscula, em letra minúscula, em negrito, itálico etc. Também denominamos Strings os blocos de texto fora de qualquer operando. Você já os viu entre aspas nos exemplos dados acima. O tratamento é o mesmo para ambas. Podemos concatená-las, pular linhas, abrir e fechar parágrafos. A seguir daremos um exemplo de declaração, outro de operação e outro de texto impresso com Strings. A declaração poderia ser assim:

```
+<tipoDeContrato> : String {
    name = "Tipo de contrato"
    request = "Insira o tipo de contrato"
    default = "Contrato de compra e venda"
}
```

A operação poderia ser assim:

```
print "\p A assinatura do " & uppercase(<tipoDeContrato>) &
      " está condicionada à entrega da documentação descrita no anexo II.\n\b"
```

Após o preenchimento, o sistema poderia imprimir o seguinte:

> 1. A assinatura do CONTRATO DE COMPRA E VENDA está condicionada à entrega da documentação descrita no anexo II.

A linha da operação ordena, basicamente, que o sistema comece um parágrafo imprimindo uma String concatenada com outra digitada pelo usuário na variável de tipo de contrato, fazendo todas as letras maiúsculas, e concatenando, por fim, com o resto do texto, antes de encerrar o parágrafo e pular uma linha. Note que toda vez que um parágrafo é aberto com '\p' ele precisa ser fechado com '\n'. Ao abrirmos um parágrafo e não fecharmos o próximo será considerado um subitem do anterior. Se o primeiro parágrafo, não fechado, for o item "1." o próximo será "1.1". Se escrevermos o próximo também sem fechá-lo o seguinte será "1.1.1" e assim por diante. É preciso ter isso em mente pois isso pode gerar muita confusão. Se, ao trabalhar com Strings, você esquecer de fechar um dos parágrafos os demais vão ficar errados. Identificar um parágrafo errado em um template grande, com diversas Strings, pode ser um desafio desagradável. Por fim, o '\b' é o comando para pular para a próxima linha sem necessariamente delimitar o parágrafo.

O "uppercase()" é um dos chamados tubes de texto, os quais veremos com mais detalhes depois. Por enquanto você deve saber declarar operandos do tipo String, imprimir e concatenar Strings, bem como organizar os parágrafos em que são impressas.

Exercício 44. Para testar a sua compreensão do tipo String, descreva a declaração, operação e como as informações ficarão no caso de uma variável de cargo de um representante legal.

Text

Text é um tipo primitivo com propriedades e comportamento idênticos aos de Strings. Assim, tudo que foi dito acima também é aplicável aqui. A única diferença é a visualização na interface. Uma variável do tipo String apresenta ao usuário um espaço pequeno, de uma linha, para que o usuário digite, já o text permite que o usuário digite e visualize um bloco inteiro de texto.

Em geral usamos String para pequenos textos com nomes, identificação de bens, complementos de endereços etc. Utilizamos text para ressalvas, cláusulas, argumentos ou quaisquer blocos inteiros de texto que o usuário queira inserir. É preciso ter em mente que ao permitirmos que o usuário insira grandes trechos de texto não temos como garantir a qualidade e a coerência do que ele insere. Podemos, por exemplo, garantir que um contrato que programamos sempre faça a concordância adequada ao número de partes (ex. as credoras ou a credora). Já o usuário que digita uma cláusula pode incorrer em alguma inconsistência ou erro. Por isso, evitamos usar muitos campos abertos, sobretudo do tipo text, e quando usamos construímos requests e helps extremamente precisos e informativos.

Exercício 45. Escreva, a seguir, como deverá ser declarado o tipo primitivo *text* para descrever o caso objeto da demanda de revisão de contrato bancário.

Boolean

O tipo **Boolean** é um tipo primitivo que se usa quando queremos guardar valores de "sim" e "não". Como vimos na Seção 4.2, um operador Booleano que admite como valores **true** e **false**, isto é, verdadeiro e falso, respectivamente. Normalmente, variáveis e campos desse tipo são usados em expressões condicionais, conforme veremos na Seção 5.7 de operações básicas. A declaração poderia ser assim:

```
1 +<querQuitarImovel> : Boolean {
2     name = "Deseja quitar o imóvel?"
3     request = "Deseja quitar o imóvel?"
4     default = false
5 }
```

É conveniente utilizar identificar variáveis e campos do tipo Boolean com palavras iniciadas com verbos. Uma operação poderia ser assim:

```
1 <querQuitarImovel> = <estaPagandoPeloImovel> AND <querPagarTodasAsParcelasDoImovel>
```

Assim como visto na álgebra Booleana, é possível realizar operações lógicas e armazená-las em variáveis ou campos do tipo Boolean. Os conectivos são escritos em caixa alta: AND (conjunção), OR (disjunção), XOR (ou exclusivo) e NOT (negação).

Exercício 46. Para testar a sua compreensão do tipo Boolean, escreva um programa que avalia a seguinte expressão Booleana: $a \vee \neg b \wedge c$.

Date e Time

O tipo **Date** e o tipo **Time** são tipos primitivos que servem para expressar datas e horários. Sendo assim, podem ser declaradas por variáveis, campos e vetores. É importante não tratar esses tipos como se fossem Strings simplesmente pois é muito comum se realizar operações com variáveis e campos destes tipos.

Há algumas funcionalidades específicas para datas e horários tais como: tomar o dia de hoje, saber se uma data (ou horário) ocorre antes ou depois de outra data (ou horário), escrever datas por extenso, extrair numericamente os anos, meses, dias, horas, minutos e segundos, etc. A seguir daremos um exemplo de declaração de data e horário, e outro de operação básicas envolvendo ambos os operandos declarados. A declaração poderia ser assim:

```
+<dataDeAssinaturaDoContrato> : Date {
    name = "Data de assinatura do contrato"
    request = "Qual a data de assinatura do presente instrumento?"
},
+<horaDaAssinaturaDoContrato> : Date {
    name = "Horário de assinatura do contrato"
    request = "Qual o horário em que o presente instrumento foi firmado?"
}
```

Uma operação comum em Lawtex é descrita a seguir:

```
<dataDeAssinaturaDoContrato> = today(),
<horaDaAssinaturaDoContrato> = now()
```

Já adiantando um pouco cenas do próximo capítulo, o código antecedente realiza uma atribuição usando duas funções específicas: **today()** que retorna o dia de hoje; e **now()** que retorna o horário local. A atribuição e as funções serão explicadas na Seção 5.7.

Exercício 47. Para testar a sua compreensão dos tipos Date e Time, responda as seguintes questões:

a) O prazo de expiração de um contrato pode ser expresso pelo tipo Date?

b) Datas de vencimento de parcelas de um pagamento podem ser guardadas em vetores de Date?

c) O ano-base para o pagamento do PIS/Pasep pode ser representado pelo tipo Date ou Time?

Integer

Integer é um o tipo primitivo correspondente aos números inteiros. É comum usarmos esse tipo para

variáveis discretas, ou seja, com informações quantitativas sobre "coisas indivisíveis". É o caso do número de filhos, número de sócios, número de garantias etc. Ninguém diria que um contrato tem meia garantia ou que uma empresa tem dois terços de sócios. Os inteiros são números, então podemos fazer cálculos com eles nas operações. Vejamos um exemplo desse tipo:

```
+<numeroDeNotificacoes> : Integer {
    name = "Notificações"
    request = "Insira quantas vezes o devedor foi notificado extrajudicialmente"
    default = 1
}
```

Exercício 48. Teste seu conhecimento a respeito desse tipo primitivo. Descreva a sintaxe de uma variável que apresente o número de dias de falta com atestado de um determinado trabalhador em uma contestação trabalhista.

Real

O **Real**, por sua vez, é o tipo dos números reais, das variáveis contínuas, ou seja, das "coisas divisíveis". Todos os comentários aplicáveis aos inteiros valem para os reais. Podemos, por exemplo, utilizar esse tipo para perguntar quantidades como litros, metros ou a porcentagem da empresa detida por um acionista:

```
+<porcentagemMajoritario> : Real {
    name = "Participação majoritária"
    request = "Indique, em porcentagem, a participação do acionista majoritário"
    default = 51
}
```

Currency

Currency está para Real assim como Text está para String. A diferença entre está apenas na visualização pelo usuário, mas as propriedades e o tratamento são os mesmos. Currency, ao contrário do Real, aparecerá para o usuário com pontos separando o milhar. No Real o único separador exibido é o de decimal com a vírgula. Essa exibição do Currency é a exibição mais comum para quantias de dinheiro, daí o nome Currency. Nada impede, contudo, que você use Currency para qualquer outro número que você queria que seja exibido com o separador de milhar. Vejamos um exemplo de Currency:

```
+<valorNominal> : Currency {
    name = "Valor nominal
    request = "Insira o valor nominal unitário de uma debênture "
    default = 1000.0
}
```

É possível definir limites numéricos para os operandos do tipo Integer, Real e Currency. Para isso, é suficiente

expressar o intervalo como argumento (uma String) da diretiva *where*, conforme é mostrado no exemplo a seguir.

```
+<valorPercentual> : Real where ("[0.0;100.0]"),
+<numeroDeParcelas> : Integer where ("[1;48]"),
+<valorDaMulta> : Currency where ("[100.0;...]"),
```

Nos dois primeiros exemplos temos restrição nas duas extremidades do intervalo. Já no segundo exemplo, temos apenas um limite inferior. Os três pontos indicam o infinito ($+\infty$).

Exercício 49. Declare abaixo o valor de um bem dado em garantia.

List

Já introduzimos o funcionamento da **List** quando discutimos a propriedade atomic. Usamos a lista para restringir a resposta do usuário à escolha dentre elementos predeterminados. Há casos em que temos que fazer isso por uma limitação do escopo. Se estivermos programando uma contestação em ação trabalhista é razoável criarmos listas de pedidos. No que tange à jornada de trabalho, por exemplo, a parte reclamante só pode discutir horas extras, intervalos, adicional noturno, horas de sobreaviso, trabalho em domingos e feriados ou a supressão de descanso semanal remunerado. Criando uma lista desses pedidos temos controle de quais argumentos devemos imprimir. Se deixarmos o usuário digitar, ele pode digitar os pedidos de infinitas maneiras, o que dificulta a avaliação. A lógica fica mais precisa e estruturada se reduzimos os pedidos a uma lista definida, a partir da qual o usuário pode estruturar as alegações que existem na inicial e nós podemos programar respostas para cada situação. Vejamos o exemplo a seguir:

```
declarations {
    +<pedidosInit> : List("Danos materiais","Danos morais","Tutela de urgência") {
        name = "Pedidos"
        request = "Indique o que a parte autora pediu"
        atomic = false
    },
    +<pedidosDef> : List("Danos materiais", "Danos morais", "Tutela de urgência") {
        name = "Pedidos"
        request = "Indique os pedidos deferidos"
        atomic = false
    }
}
operations {
    <pedidosInicial>.ask(),
    <pedidosDeferidos>.options = <pedidosInicial>,
    <pedidosDeferidos>.ask()
}
```

Vamos entender o que essas linhas de código fizeram. Em primeiro lugar, vemos que no plano das declarações

há duas variáveis com as mesmas opções. A priori tudo que pode ser pedido a um juiz pode ser deferido por um juiz. O que é passível de deferimento deve responder dinamicamente ao que foi pedido. Então, na parte operativa, vemos que perguntamos ao usuário os pedidos da inicial. Em seguida mudamos a propriedade options. Da variável de pedidos deferidos. Agora serão mostradas como opções passíveis de serem escolhidas apenas aquelas que foram selecionadas na pergunta anterior. Agora que restringimos as opções da segunda variável ela está pronta para ser perguntada ao usuário. Ao realizar esse procedimento é importante lembrar que uma lista nunca pode ter suas opções ampliadas, só restritas. Assim, se você declarou, digamos, cinco opções para uma lista, ao longo da execução o máximo que você poderá fazer é restringir essas opções a um subconjunto delas.

Por fim, vale adiantar um outro aspecto da operação com listas não atômicas. Como as listas não atômicas podem ser respondidas com mais de uma opção, não avaliamos se elas são iguais a algo, mas sim se esse algo pertence ao conjunto de respostas selecionadas. Imagine que o usuário elencou dentre os pedidos horas extras, descanso semanal remunerado e adicional noturno. Nesse caso a avaliação

<pedidosJornada> == "Horas extras"

não faria sentido, pois estamos perguntando se uma lista com vários elementos é igual a uma String. Se a lista fosse atômica e só pudesse ter uma opção, teríamos, na prática, sempre uma única String como resultado, de forma que a avaliação faria sentido. Nesse caso, contudo, a avaliação correta é

"Horas extras" IN <pedidosJornada>,

ou seja, não avaliamos se os pedidos são iguais a horas extras, mas sim se a opção horas extras está entre os pedidos.

Exercício 50. De acordo com o exemplo acima indicado, faça a declaração de uma lista de verbas rescisórias trabalhistas que poderão constar em uma contestação trabalhista.

Uma ressalva tem de ser feita com relação à lista não atômica. Ela tem características de conjunto, mas ela não é tratada pelo sistema efetivamente como um conjunto. Não é possível, por exemplo, aplicar filtros ou o foreach sobre ela. Para usar as ferramentas aplicáveis aos conjuntos, no caso os vetores, é preciso atribuir a lista a um vetor de Strings. Isso vai ficar mais claro ao fim do material, depois que você já conhecer essas funcionalidades.

As opções de uma lista podem ser alteradas ao longo da execução. Suponha que, na automação de uma sentença, eu pergunte ao usuário quais foram os pedidos da parte autora e, em seguida, pergunte quais pedidos serão deferidos. A segunda lista é uma versão restrita da primeira, pois o juiz não vai deferir um pedido que não foi feito. Portanto as opções de uma lista são uma propriedade, assim como name, request, atomic etc.

As listas têm um comportamento muito semelhante ao das Strings em seu uso, mas restringir o preenchimento do usuário à escolha de uma opção predefinida tem suas particularidades. A primeira delas é a definição da propriedade atomic já descrita. Essa propriedade determina se o usuário deve escolher apenas uma das opções, ou seja, se elas são excludentes entre si, ou se ele pode escolher mais de uma simultaneamente. Como vimos atomic verdadeiro implica que as opções são excludentes e falso que mais de uma opção pode

ser escolhida ao mesmo tempo.

Imagine que definimos uma lista com 100 elementos. É muito inconveniente dar Ctrl 'C' e Ctrl 'V' nessas listas a todo momento. O custo de manutenção é altíssimo. Existe a possibilidade de evitar esse despautério e declarar uma lista uma só vez, e após isso, instanciá-la em outras variáveis, isto é, criar variáveis com esse tipo. As declarações de lista seguem a seguinte sintaxe:

```
declarations {
    *list[EstadosDoSudeste] {
        name = "Lista de estados do Sudeste"
        options = ("Espírito Santo", "Minas Gerais", "Rio de Janeiro", "São Paulo")
        type = "String"
    }
}
```

Sempre o identificador de um tipo, e no caso específico, a lista, deve iniciar com letra maiúscula. Em Lawtex, além de criar listas como a descrita no exemplo anterior, você pode ir além disso e criar listas associativas estruturadas. Com essas listas, conseguimos relacionar informações entre si, conforme o exemplo abaixo.

```
declarations {
    *list[EstadosSiglasCapitaisDoSudeste] {
        name = "Lista associativa de estados do Sudeste"
        fields = {"nome", "sigla", "capital"}
        options = ({"Espírito Santo", "ES", "Vitória"},
                   {"Minas Gerais", "MG", "Belo Horizonte"},
                   {"Rio de Janeiro", "RJ", "Rio de Janeiro"},
                   {"São Paulo", "SP", "São Paulo"})
        type = "String"
    }
}
```

O exemplo acima, nos permite associar respostas do usuário a outras informações implícitas, como por exemplo, a partir do estado respondido obter a capital e a sigla do mesmo. O asterisco indica que a lista declarado pode ser usada por mais de uma variável (lista global). O conceito de global e local será melhor explorado em seções posteriores.

Document

O tipo **Document** permite o desenvolvimento de templates muito interessantes. Esse tipo indica que essa variável não é uma informação ou um objeto com um bloco de informações, mas sim um template inteiro. Esse tipo permite que um template gere documentos que buscam informações de outros documentos gerados a partir de um template especificado. É o caso dos instrumentos de garantia que usam informações do contrato principal, do recurso que usa informações da contestação, do aditamento que usa as informações de um contrato e assim por diante. Quando o usuário for responder essa pergunta, ele verá uma lista dos documentos (feitos a partir do template especificado) disponíveis para sua conta. Basta escolher um deles para que suas informações sejam aproveitadas. A única ressalva que deve ser feita é que quando você for

utilizar a informação de outro documento no seu template você deve indicar em que ponto do documento está a informação desejada, pois ele terá sua própria topologia e organização. Vejamos um exemplo:

```
1  declarations {
2      +<contestacao> : Document[TEMP_Contestacao] {
3          name = "Contestação"
4          request = "Selecione a contestação protocolada neste processo"
5      }
6  }
7  operations {
8      if ("Perícia" IN <contestacao$Metainfo:pedidos.producaoProvas>) {
9          print "\pNa contestação requereu-se a produção de perícia sobre a assinatura
10                 discutida no presente caso.\n\b",
11         if ( <contestacao$Metainfo:pericia.pediuPericiaGrafoTecnica> ) {
12             print "\pFoi pedido perícia grafotécnica entre a assinatura da procuração
13                    e do contrato discutido.\n\b"
14         }
15     }
16 }
```

No exemplo declaramos, em primeiro lugar, que uma das variáveis utilizadas vem de outro documento, no caso, uma contestação. No exemplo da Linha 8, podemos observar uma comparação envolvendo o campo **producaoProvas** pertence ao objeto **pedidos**, que por sua vez foi declarado no **metainfo** do template **TEMP_Contestacao**. Analogamente, na Linha 11, encontramos o campo **pediuPericiaGrafoTecnica** do objeto **pericia**, também declarado no **metainfo** do template **TEMP_Contestacao**. Em ambos os casos, avaliamos os valores dos campos de outro documento, para imprimirmos os textos desejados. Essa funcionalidade poupa o usuário de ler a contestação e inserir novamente informações já inseridas no passado.

Existe ainda uma possibilidade de nem exibir na tela o seletor do documento. Para isso, use a propriedade default para nomear o documento que contém as informações a serem consultadas e declare a variável invisível. Assim, conseguimos criar um ecossistema onde documentos se comunicam e se gerenciam de certa forma.

Struct

Estruturas, ou Structs, são tipos de dados que agrupam outros subtipos. São basicamente *containers* de operandos. Objetos, campos e vetores podem ser do tipo Struct. Essa informação é extremamente relevante e frequentemente esquecida. Struct é um tipo. Uma estrutura é vista também como uma classe de objetos. Quando você declara uma Struct você define certas informações e propriedades que definem uma categoria, sendo que o que vai na parte operativa do código não é a própria estrutura mais variáveis daquela categoria. As estruturas globais, por exemplo, são tipos reaproveitáveis, já um objeto é um exemplo concreto.

Retomando um exemplo já utilizado, posso declarar uma Struct de pessoa e depois declarar diversos objetos desse tipo. Posso, por exemplo, criar uma Struct chamada de Parte, e a partir dela, **instanciar** uma parte credora, uma parte devedora e uma série de partes garantidoras, ou seja, duas variáveis e um vetor de pessoas. No caso de uma Struct, a instanciação é a concretização de seu objeto no ato da declaração. Esse exemplo é codificado abaixo:

```
declarations {
    *struct[Parte] {
        name = "Dados da parte"
        request = "Insira os dados da parte"
        help = "São os sujeitos do contrato."
        fields {
            +[nome] : String {
                name = "Nome da parte"
            },
            +[genero] : List ("Masculino", "Feminino") {
                name = "Gênero"
                request = "Indique o gênero da parte"
                atomic = true
            }
        }
    },
    +<parteCredora> : *Parte,
    +<parteDevedora> : *Parte,
    +|partesTerceiras| : Vector[*Parte]
}
```

Vejamos agora o que cada trecho de código significa. Em primeiro lugar, indicamos que estamos declarando uma Struct. Sempre que inserirmos o asterisco (∗) antes da declaração estamos indicando que ela é global. É frequente criar uma estrutura e usá-la só uma vez. Neste caso podemos declará-la *inline*, ou seja, após os dois pontos da declaração do objeto. Já as estruturas usadas múltiplas vezes são as estruturas globais, que ficam acessíveis a qualquer engenheiro ou engenheira sem a necessidade de declará-las. Em geral, as componentes globais podem ser usados por outros engenheiros sem cópia de código. Discutiremos mais da mecânica dos componentes globais em tópicos seguintes, mas por enquanto basta saber que podemos usar componentes globais que foram declarados em outros documentos ou arquivos. Declarando a estrutura com esse asterisco nos permitimos que outros programadores usem essa estrutura ou que nós mesmos utilizemos em mais de um contexto, mais de uma vez. No código acima, o asterisco define que a estrutura *Parte* é global.

Em seguida, vemos as propriedades da estrutura. Cada campo da estrutura tem suas propriedades, mas há propriedades que se aplicam ao *card* todo. É o caso do name, request e help. Em geral essas propriedades são instruções gerais sobre o objeto que será caracterizado. Os fields (ou campos), por sua vez, indicam os operandos que caracterizam uma estrutura.

Relembrando: Não confunda Struct com objeto. O usuário preenche o *card* de um objeto do tipo Struct, não a Struct em si. Daí a diferença na sintaxe de campos de estruturas e de variáveis.

Outro aspecto importante das estruturas é que elas permitem um uso atípico de operações em uma parte eminentemente declarativa. As estruturas permitem a existência de ifstructs e de loaders. Pode parecer confuso, mas há diversas boas razões para tanto. Embora as operações nas estruturas sejam muito semelhantes às operações no código seu uso tem um objetivo diferente. Em geral de ifstructs e loaders são operações relacionadas à exibição de perguntas para o usuário, ao passo que as operações da parte tipicamente ope-

rativa do código são efetivamente as lógicas e impressões que compõem o documento. Um exemplo claro disso é o CPF e o CNPJ. Se o usuário respondeu que o sujeito em questão é uma pessoa física, faz sentido pedirmos o campo CPF e nunca mostrarmos o campo CNPJ. Mostrar todas as informações possíveis e deixar o usuário decidir qual é aplicável e qual não é só dificultaria o uso. Logo, a Struct permite operações para adequarmos as perguntas às respostas anteriormente dadas pelo usuário. Veremos isso com mais detalhes no tópico dedicado exclusivamente às operações dentro de Structs. Seguimos com outro exemplo de estrutura:

```
+<emprestimoCCB> : struct[EmprestimoCCB] {
    name = "Dados do empréstimo"
    request = "Insira os dados do empréstimo"
    help = "Esse card só vale para empréstimos feitos por meio de CCB"
    id = "R$ " & [valorFinanciado]
    fields {
        +[numeroDeParcelas] : Integer {
            name = "Número de parcelas"
            request = "Indique o número de parcelas"
        },
        +[numeroCedula] : String {
            name = "NúmerovalorFinanciado da CCB"
            request = "Insira o número da CCB"
        },
        +[valorFinanciado] : Currency {
            name = "Valor financiado (em R$)"
            request = "Insira o valor financiado em reais"
        },
        +[valorParcela] : Currency {
            name = "Valor das parcelas"
            request = "Insira o valor das parcelas"
        },
        +[dataEmissao] : Date {
            name = "Data da emissão"
            request = "Insira a data de emissão da CCB"
        },
        +[localEmissao] : *Logradouro,
        +[status] : List("Em atraso","Em dia","Quitado") {
            name = "Status"
            request = "Indique o status do pagamento"
            atomic = true
        }
    }
}
```

No exemplo anterior, temos uma definição de estrutura *inline* cujas três propriedades da Struct (name, request e help) estão pedindo que o usuário insira os dados do empréstimo, deixando claro que essa estrutura se aplica apenas a empréstimos feitos por meio de uma cédula de crédito bancário ("CCB"). Em seguida, vemos algo que ainda não foi discutido, o id. Ele é o identificador da estrutura. Essa identificação tem duas

funções. A primeira é permitir que possamos imprimir o objeto. Se digitamos

print <emprestimoCCB>

seria de se esperar que o sistema não soubesse o que imprimir, pois <emprestimoCCB> não é uma informação, mas um complexo de informações, com número de parcelas, calores, datas, locais, entre outras informações. Se definimos, contudo, um id o sistema aceitará essa lógica imprimindo a informação que definimos como identificador daquele objeto. Se definimos que o id da Struct *EmprestimoCCB é seu <valorFinanciado>, a impressão de <emprestimoCCB> será o valor financiado acompanhado por R$.

Além disso, o identificador viabiliza a funcionalidade select. Essa funcionalidade permite criar listas em que o usuário escolhe um ou mais elementos de um vetor. Veja que essa é uma lista atípica porque ela não é a escolha de uma ou mais opções de texto previamente definidas. Na lista do select o usuário pode se deparar com os elementos do vetor que ele mesmo preencheu. Um exemplo disso é quando pedimos para o usuário inserir os advogados que assinam a peça e, depois, que escolha um deles para receber notificações. Nesse ponto o id é essencial pois é ele que vai identificar cada elemento na lista. Como o usuário vai escolher um elemento de um conjunto de objetos complexos, se apresentássemos a lista com nome, OAB, endereço etc. de cada um a lista não seria agradável. Assim, quando o sistema tem que transformar um conjunto de objetos complexos em uma lista, ele o faz listando pelo identificador. Assim, se na estrutura de advogados definimos como identificador o nome, o usuário verá uma lista com o nome dos advogados que ele pode escolher. Voltando ao exemplo acima, colocamos como identificador o valor financiado. O pressuposto é que o usuário identifica cada empréstimo pelo seu valor, mas outro candidato forte a identificador seria o número da CCB. Essa é uma questão de usabilidade que depende do conhecimento de quem usará o template.

Em seguida nos deparamos com os fields, ou seja, o conjunto de informações que caracterizam objetos complexos dessa categoria. No caso, as informações associadas a um empréstimo feito por meio de CCB. Não vamos tratar de cada campo pois você já deve estar acostumado com a sintaxe da maior parte deles. Vamos discutir aquilo que ainda não foi estudado nesse material. Veja que o campo local de emissão é do tipo Struct. Pode parecer surpreendente imaginar uma Struct dentro de Struct, mas na verdade é bem comum. O criador do Lawtex, aliás, sempre ressalta que o sistema permite Struct dentro de Struct dentro de Struct dentro de Struct e assim por diante. Imagine uma estrutura de sociedade anônima. Essa sociedade é um objeto complexo e tem seus dados, tais como CNPJ, sua data de constituição etc. Um de seus dados é a composição de sua diretoria. Cada um desses diretores é, por sua vez, uma pessoa natural, um objeto complexo com nome, CPF, RG etc. Cada um desses diretores pode ter como uma das informações que os definem quem é seu procurador. Procurador, mais uma vez, é um objeto complexo com nome, CPF, RG, endereço etc. Ocorre que endereço é um objeto complexo, com rua, número, complemento, CEP, cidade etc. Logo tanto a sociedade anônima, quanto seus diretores e procuradores de seus diretores têm, como um dos campos que os definem, um objeto complexo chamado endereço. Aliás, conectando essa discussão com a anterior, para facilitar o uso da informação endereço e evitar sintaxes longas como

<sociedadeEmissora.presidente.procurador.endereco.rua>

deixamos como identificador do endereço sua qualificação, de forma que, para imprimir o endereço do procurador nesse caso bastaria utilizar

print <sociedadeEmissora.presidente.procurador.endereco>,

se você estiver utilizando os componentes globais da Looplex. Os pontos que você vê na sintaxe acima são a maneira de informar ao sistema que estamos acessando uma informação dentro de uma estrutura anterior.

Voltando ao exemplo acima, o local de emissão da CCB é um endereço, ou seja, um objeto complexo nos termos que acabamos de definir.

Herança: Herança é um conceito emprestado de linguagens orientadas a objeto que nos permite estender uma estrutura, criando uma especialização (ver Seção 4.3). Com ela, é possível criar associações de paternidade entre classes por meio da diretiva **extends**. Ele permite que você informe ao sistema que quer criar uma nova estrutura, com todas as características de outra, acrescentando ou redefinindo apenas as modificações que você deseja. Isso é útil para não termos que reescrever a estrutura toda. É o caso, por exemplo, de uma pessoa natural capaz, que herda suas propriedades de pessoa natural que, por sua vez herda suas propriedades de sujeito e assim por diante. Abaixo segue um exemplo:

```
*struct[Sujeito] {
    fields {
        +[nome] : String {
            name = "Nome ou denominação"
        },
        +[genero] : List ("Masculino", "Feminino") {
            name = "Gênero"
            atomic = true
        }
    }
}
*struct[PessoaNatural] extends *Sujeito {
    fields {
        +[nome] : String {
            name = "Nome da pessoa"
        },
        +[nacionalidade] : String {
            name = "Nacionalidade"
        },
        +|profissoes| : Vector[String] {
            name = "Profissões"
        },
        +[estadoCivil] : String {
            name = "Estado civil"
                                }
    }
}
```

No exemplo acima, a Struct *PessoaNatural possui, ainda que implicitamente, todos os fields da estrutura Pai, a saber, a Struct *Sujeito, além dos quatro campos declarados. Ao todo, são cinco campos: [nome], [genero], [nacionalidade], |profissoes| e [estadoCivil]. Note que o campo [nome] foi declarado tanto na estrutura sujeito quanto na estrutura pessoa natural. Nesses casos, prevalecem os campos declarados na estrutura filha, e os campos repetidos na estrutura pai, deixam de existir. Como consequência, a propriedade **name** do campo [nome] da Struct *PessoaNatural é “Nome da pessoa”.

Pré-loaders: Ao fazermos heranças podemos fazer pré-loaders, ou seja, carregamos alguns campos com valores e propriedades. Se estamos, por exemplo, criando uma estrutura de emissora de debêntures sabemos

que ela será sociedade por ações, de forma que todas as perguntas sobre tipo de sujeito, tipo de pessoa jurídica e tipo societário podem ser respondidos desde logo, pois a estrutura e seu contexto já têm um escopo restrito. A seguir, vejamos um exemplo de pré-loader.

```
+<localSujeito> : struct[LocalSujeito] extends *SujeitoQualificado where (
        [papel] = "",
        [nome] = "Almeida Prado Ltda.",
        [tipoDeSujeito] = "Pessoa jurídica",
        [tipoDePj]= "Sociedade",
        [subtipoDePj] = "Sociedade empresária limitada",
        [cnpj] = "12.946.691/0001-29",
        [genero].mandatory = "+",
        [genero].name = "Gênero da Sociedade",
        [genero].request = "Indique o gênero",
        [genero].default = "Feminino",
        |logradouros|.mandatory = "+",
        |logradouros|.upper = 1,
        |logradouros|.lower = 1,
        [logradouros{0}.tipo] = "Sede",
        [logradouros{0}.pais] = "Brasil",
        [logradouros{0}.principal]= "Avenida Liberdade",
        [logradouros{0}.numero] = "1485",
        [logradouros{0}.complemento] = "15° andar",
        [logradouros{0}.bairro] = "Liberdade",
        [logradouros{0}.codigoPostal] = "01503-010",
        [logradouros{0}.uf] = "São Paulo",
        [logradouros{0}.cidade] = "São Paulo" ) {
    name = "Sociedade"
    request = "Insira os dados da sociedade"
}
```

No exemplo anterior, percebemos os pre-loaders determinando valores de **campos** e **propriedades**, senda estas definições feitas todas dentro da cláusula **where**. Os pré-loaders são ferramentas poderosas e essenciais em grandes projetos de templates, pois além de economizar tempo de preenchimento, trazem mais segurança no preenchimento consistente via Lawtex.

Quanto à herança, supondo que tanto uma estrutura pai quanto uma estrutura filha tenham pré-loaders, os pré-loaders da pai são carregados antes dos pré-loaders da filha. Sendo assim, os pré-loaders da estrutura filha sobrepõem os da estrutura pai caso hajam conflitos.

Exercício 51. A struct é um tipo muito utilizado em templates pelos engenheiros e engenheiras, por isso, vamos testar o conhecimento. Descreva a declaração de endereço de determinada pessoa.

5.7 Operações e Comandos

Print

O **print** é a operação mais comum. Essa operação já apareceu em diversos dos exemplos acima. Seu papel fundamental é estabelecer que o computador escreva algo no documento final, ou seja, que ele imprima algo. Essa impressão pode ser modificada e personalizada com os tubes de texto que veremos em tópicos seguintes, mas ela deve existir antes da aplicação de qualquer um deles.

Há três aspectos muito importantes dessa operação. A primeira é que ela tem uma sintaxe mais complexa e outra simplificada. A sintaxe simples é a que vimos nos exemplos, a diretiva **print** associada ao que queremos imprimir. A sintaxe mais complexa deve ser utilizada, contudo, se você quer fazer mais de uma versão do mesmo texto. Isso porque o sistema permite mais de uma forma de redação, quando essas redações todas funcionariam no mesmo ponto do código, do ponto de vista lógico. Imagine que eu tenha cinco formas de dizer que o contrato será por prazo indeterminado. Imagine, agora, um ponto do código em que garantimos que

<prazo.tipo> == "Indeterminado".

Nesse ponto poderíamos imprimir qualquer uma das cinco alternativas. Logo podemos usar a sintaxe mais longa e já colocar as cinco alternativas lá. Poderemos acionar cada uma delas por escolha nossa ou pela escolha do usuário. Podemos deixar que usuário use cada uma e veja qual prefere. Podemos redigir cada uma delas com viés a favor de um ou de outro contratante e operar esse viés com base em quem pediu a minuta.

O segundo aspecto é que podemos inserir de forma abreviada uma condição no texto se essa condição afeta especificamente a impressão. Vejamos o exemplo a seguir:

```
1 print "\pRespondem solidariamente pelas obrigações assumidas " &
2       printIf(<tipoDeContrato> == "Escritura", "nessa","nesse") &
3       <tipoDeContrato> & ":\n\b"
```

Aqui precisávamos de um pequeno ajuste no texto, pois para escrituras precisamos de feminino e para todos os outros contratos, instrumentos, termos acordos etc. precisamos de masculino. Daí inserir o printIf. O printIf, é uma forma resumida de fazer uma condição quando ele só trata de texto, daí termos, entre parênteses, a condição, em seguida, separado por vírgula, o texto a ser impresso quando a condição é verdadeira e, por último e também separado por vírgula, o texto a ser impresso quando a condição é falsa.

Por fim, o terceiro aspecto importante da impressão é a possibilidade de aplicação da gramática. Para concordar palavras com vetores de sujeitos ou variáveis cujo gênero gramatical sempre é pedido ao usuário, o sistema automaticamente faz a concordância de gênero e número se indicarmos qual é o vetor ou variável de referência. Ele é capaz de escolher, por exemplo, a opção correta dentre “a autora requer”, “o autor requer”, “as autoras requerem” e “os autores requerem”, de acordo com as informações que o usuário selecionou no sistema, por exemplo o número de autores e o gênero de cada autor. Para outras questões gramaticais que não envolvem sujeitos, a solução mais rápida é o printIf.

A gramática é, contudo, uma funcionalidade poderosa do sistema. Além do vetor ou variável de referência seu uso exige que o engenheiro ou a engenheira se certifique de que a palavra a ser concordada está no

dicionário da Looplex.

A manutenção da gramática é atualmente feita pela Looplex mediante a alimentação de uma base gramatical. Dessa forma, infelizmente se a palavra não estiver no dicionário é preciso acrescentá-la ou a gramática não vai funcionar. O uso da gramática exige, atenção redobrada! Além de se certificar de que a palavra exista no dicionário Looplex, o engenheiro ou a engenheira deverá inserir a palavra em sua conjugação singular-masculina, com a mesma grafia que se encontra no referido dicionário, por exemplo, respeitando as letras minúsculas e maiúsculas, dentre outros, para que o sistema reconheça a palavra ou expressão, caso contrário a funcionalidade não será executada.

A sintaxe da gramática para concordância de gênero é a seguinte:

```
<parteAutora>.grammar("solteiro")
```

Ou, para concordância de gênero e número com vetores:

```
|partesDevedoras|.grammar("qualificado")
```

Atribuições

As atribuições são muito comuns nos templates. Uma atribuição ocorre quando a informação inserida num operando é definida no código e não pelo usuário. Vamos retomar um exemplo já fornecido para entender o conceito:

```
declarations {
    +<numeroDeParcelas> : Integer {
        name = "Número do parcelas"
        request = "Insira o número de parcelas"
    },
    +<valorDaParcela> : Currency{
        name = "Valor da parcela"
        request = "Insira o valor individual de cada parcela"
    },
    -<valorTotalDoContrato> : Currency
}
operations {
    <valorTotalDoContrato> = <numeroDeParcelas> * <valorDaParcela>
}
```

Nesse caso não precisávamos pedir ao usuário o valor total do contrato, pois ele podia ser preenchido por atribuição. Um sinal de igual em Lawtex significa uma atribuição. É diferente de dois sinais de igual que indicam uma avaliação se duas coisas são iguais. Lemos a linha acima como "valor total do contrato recebe o número de parcelas multiplicado pelo valor da parcela. É importante ler dessa forma para facilitar a compreensão de linhas como a seguinte:

```
<valor> = <valor > + <valor> * <multa>
```

Em termos matemáticos essa linha só faria sentido se valor fosse igual a zero ou se a multa fosse igual a zero. Afinal, subtraindo "<valor>" dos dois lados teríamos "<valor> * <multa>" igual a zero. Ocorre que nem valor nem multa devem ser zero. Justamente porque esse sinal de igual deve ser lido como uma atribuição. Essa linha deve ser lida como "a variável 'valor' recebe seu 'valor original' acrescido de seu 'valor multiplicado pela multa'. Trata-se de uma instrução, não de uma afirmação matemática. A atribuição vale para operandos de qualquer tipo, respeitados os métodos adequados ao tipo (uma String, por exemplo, não pode ser somada com '+', mas sim concatenada com '&').

As modificações de propriedade que mencionamos acima nada mais são do que atribuições realizadas nas propriedades. Podemos, inclusive, fazer atribuições em posições de vetor. Imagine uma ação que discute um contrato. Nela, perguntamos quem é a parte autora e se ela é parte no contrato. Se o usuário responder que sim sabemos que ela é parte, mas pode haver outras. Assim, atribuímos ela como primeira posição no vetor de partes do contrato discutido e trazemos o vetor para que o usuário insira as demais partes. Ele já verá a parte autora como primeira posição do vetor, sabendo que a atribuição ocorreu. Esse comportamento, contudo, não vale sempre. A atribuição feita a uma variável faz com que ela não seja exibida para o usuário. A atribuição só é exibida se ela ocorrer na posição de um vetor ou no campo de uma estrutura.

Por fim, as atribuições não são modificáveis pelo usuário. Elas ficam em cinza quando exibidas na tela e bloqueadas para alterações.

Exercício 52. Agora vamos aperfeiçoar seu conhecimento sobre atribuições. Descreva abaixo a atribuição referente ao valor nominal de uma emissão pública de notas promissórias.

Modificações de propriedades: Como já destacamos nos itens sobre propriedades e nos itens sobre atribuições, podemos modificar as propriedades de operandos a qualquer momento. Podemos restringir as opções de listas, modificar limites mínimos e máximos de vetores, modificar perguntas e nomes de variáveis etc. Para fazer a referência a uma propriedade usamos ponto. Note que o ponto dentro da variável indica o acesso a campos de um objeto, ao passo que o ponto fora dela é usado para acessar ou alterar propriedades. Vejamos dois exemplos:

```
1    <papel>.request = "Insira a posição de " & <autor.nome> &
2                      " no contrato discutido na ação",
3    <papel>.options = {"Parte devedora principal", "Parte devedora solidária",
4                      "Parte garantidora"}
```

No exemplo adaptamos a pergunta do papel da parte autora no contrato colocando seu nome na pergunta. Tal operação não é logicamente imprescindível. O usuário sabe quem é a parte autora, mas identificá-la especificamente melhora a usabilidade do template. Além disso, inserimos no exemplo o acesso a um campo da estrutura de autor para diferenciar o uso do ponto na modificação da propriedade. Por fim, também modificamos as opções para ilustrar a sintaxe dessa operação.

Cálculos

Os cálculos também já foram exemplificados acima. A notação dos cálculos em Lawtex é muito parecida com

a utilizada em planilhas. Multiplicações são indicadas por '∗', divisões por '/', adições por '+', subtrações por '−' e potenciação por '^'. Lembre-se que a raiz nada mais é do que um expoente fracionário, de forma que podemos sempre expressá-las usando notação de potenciação. Vejamos como a fórmula da taxa de juros composta é expressa em Lawtex:

```
<valorFinal> = <valorInicial>*(1+<taxaDeJuros>)^<prazo>
```

Lembre-se de adequar os tipos das variáveis ao seu papel nos cálculos. O prazo, por exemplo, pode ser um inteiro, mas o valor final esperamos que seja um valor Real ou do tipo Currency. Lembre-se também de garantir que no momento em que os cálculos são executados, todas as informações estão disponíveis (já foram perguntadas aos usuários), que não há divisões por zero, raízes pares de números negativos etc.

If, elseif e else

Já vimos diversos exemplos de "if", "elseif" e "else" ao longo desse material. Usamos o if quando queremos restringir uma operação a determinada condição. Quando usamos o if estamos impondo que as linhas de código contidas entre as chaves só sejam executadas se a condição entre parênteses for verdadeira. Se inserimos um "elseif" estamos impondo como condição que (1) o previsto no if seja falso e (2) que o previsto no elseif seja verdadeiro. Trata-se de uma alternativa restrita. Isso porque o else é uma alternativa irrestrita: para qualquer outro caso que não o imposto no if será executado o disposto no else.

Essas distinções parecem simples, mas geram confusão com frequência. É comum colocarmos por engano no elseif uma condição que não é excludente da prevista no if. Também é comum colocarmos no else linhas que não queríamos que fossem executadas em todos os casos excluindo o previsto no if. Como regra de bolso faça vários ifs separados se você aceita que talvez essas condições se manifestem em conjunto e use elseif apenas se você quiser que as condições sejam mutuamente excludentes. Seguiremos com um exemplo de como **NÃO** fazer condições em Lawtex:

```
if (EXISTS <devedor> IN |devedores| :
            <devedor.tipoDeSujeito> == "Pessoa natural" AND
            <devedor.estadoCivil> == "Casado") {
    use *topic[TOP_Rep_Casado]
} elseif (EXISTS <devedor> IN |devedores| :
                  <devedor.tipoDeSujeito> == "Pessoa jurídica" AND
                  <devedor.tipoDePJ> == "Sociedade Anônima") {
    use *topic[TOP_Rep_SA]
} else {
    use *topic[TOP_Rep_LTDA]
}
```

A sintaxe do "if", "elseif" e "else" está certa. O erro desse trecho é lógico. Em primeiro lugar, observe que a condição utilizada é de existência. Veremos com mais detalhes essa condição, mas você deve ler, por exemplo, a primeira condição da seguinte forma: "se existir pelo menos um devedor qualquer, dentre o conjunto de devedores, que é pessoa natural e que é casado". A segunda condição deve ser lida como "se existir pelo menos um devedor qualquer, dentre o conjunto de devedores, que é pessoa jurídica e que é sociedade anônima". Dentro das condições estamos usando um código modularizado global, que são cláusulas de representações

de pessoas casadas (dizendo, por exemplo, que as outorgas uxórias necessárias foram concedidas etc.), de representações de S.A. (dizendo, por exemplo, que a celebração do contrato foi autorizada nos termos do estatuto, assembleia geral etc.) e de representações de limitada (dizendo, por exemplo, que a celebração foi autorizada nos termos do contrato social, reunião de sócios etc.).

No fato descrito acima, vemos o primeiro erro lógico. É possível que haja, dentre os devedores, uma pessoa física, uma S.A. e uma limitada. Nesse caso seria necessário inserir três representações diferentes. Ocorre que o código não vai fazer isso. Se ele verificar uma pessoa natural casada no conjunto, ele só vai imprimir a representação de pessoas casadas e parar. Se não houver pessoas naturais casadas, mas houver S.A. e limitada, ele, seguindo a ordem do código, vai imprimir a representação de S.A. e parar.

Por fim, há um erro lógico de deixar a representação de limitada no else: Se não houver pessoa natural casada ou S.A. ele vai automaticamente imprimir a representação de limitada, sendo que o conjunto pode ser inteiro composto por cooperativas, sociedades de advogados ou fundações. Essa lógica não só deixa de imprimir cláusulas necessárias, como pode imprimir cláusulas incompatíveis com os dados informados pelo usuário! Um perigo, de fato!

Vejamos agora a lógica correta:

```
if (EXISTS <devedor> IN |devedores| :
            <devedor.tipoDeSujeito> == "Pessoa natural" AND
            <devedor.estadoCivil> == "Casado") {
    use *topic[TOP_Rep_Casado]
},
if (EXISTS <devedor> IN |devedores| :
            <devedor.tipoDeSujeito> == "Pessoa jurídica" AND
            <devedor.tipoDePJ> == "Sociedade Anônima") {
    use *topic[TOP_Rep_SA]
},
if (EXISTS <devedor> IN |devedores| :
            <devedor.tipoDeSujeito> == "Pessoa jurídica" AND
            <devedor.tipoDePJ> == "Sociedade Limitada") {
    use *topic[TOP_Rep_LTDA]
}
```

Agora sim temos condições não excludentes e restringimos o uso da representação de limitada à situação em que há pelo menos uma limitada no conjunto.

Exercício 53. Agora vamos praticar! Descreva a sintaxe de operação no caso de 3 credores predefinidos (João Credor, Maria Credora e João Maria Credor), inserido o endereço de cada credor.

Pertinência

Já indicamos como avaliar se algo pertence a algum conjunto. Vimos essa sintaxe quando discutimos listas

e quando discutimos a propriedade atomic. Também vimos essa sintaxe logo cima, no tópico sobre if, elseif e else. A pertinência em Lawtex é expressada pela sintaxe IN. Com ela podemos avaliar se determinado objeto pertence a um vetor e se determinado texto pertence ao conjunto de respostas do usuário numa lista não atômica. Ele pode ser usado de maneira direta, como por exemplo

if ("Horas extras" IN <pedidos>),

ou em um EXISTS (ou FORALL) conforme se vê no tópico anterior. A primeira avaliação é mais simples, pois o sistema apenas busca o texto pedido nas respostas do usuário. Na segunda o sistema busca num conjunto de objetos complexos qualquer elemento que tenha a característica desejada.

Veja que a avaliação de pertinência não conta o número de elementos de um vetor que têm determinada característica. Para saber se existe algo com a característica ou contar quantos são esses elementos, é preciso outra lógica, que estudaremos adiante. Por enquanto tenha em mente que essa sintaxe é apropriada para conjuntos, que em Lawtex são as listas e os vetores. Por fim, você também verá que na sintaxe foreach o IN também desempenha esse papel.

Quantificadores existenciais e universais

Já utilizamos o **EXISTS** diversas vezes nos exemplos acima. Nos adiantamos dessa forma simplesmente porque seu uso é o que gera os maiores desafios do ponto de vista dos operadores lógicos e da pertinência. Tentamos usar os exemplos mais difíceis, não por sadismo (ok, talvez por um pouco de sadismo), mas porque dominando os casos mais complexos, resolver os simples é trivial.

O EXISTS é a forma de avaliar se, em determinado conjunto, existe um ou mais elementos que satisfazem determinada condição. Ele não conta quantos satisfazem, apenas retorna "verdadeiro" se existir um ou mais elementos com aquela característica e "falso" se não existir nenhum elemento com aquelas características. Vejamos um exemplo:

```
1 if (EXISTS <devedor> IN |devedores| : <devedor.certidaoTrabalhista> == "Positiva")
```

Um jeito de ler essa linha é "se existir qualquer devedor no conjunto de devedores tal que sua certidão perante a justiça do trabalho seja positiva". Essa é uma boa condição, por exemplo, para alertar o usuário ou para pedir mais garantias, já que existe um ou mais devedores com passivo trabalhista.

O irmão do EXISTS é o FORALL. O FORALL é a forma de avaliar se, em determinado conjunto, todos os seus elementos satisfazem determinada condição. Ele retorna "verdadeiro" se todos os elementos possuem aquela característica e "falso" se existir algum elemento que não satisfaz aquelas características. Vejamos um exemplo:

```
1 if (FORALL <devedor> IN |devedores| : <devedor.certidaoTrabalhista> == "Positiva")
```

Um jeito de ler essa linha é "se todo devedor no conjunto de devedores tal que sua certidão perante a justiça do trabalho seja positiva".

Contagem e filtros

Como acabamos de descrever, a avaliação de existência não faz contagens. Para contar quantos elementos satisfazem uma condição é frequente usarmos o foreach. Da mesma forma que, no exemplo acima, pedíamos que o sistema imprimisse um texto para cada elemento, aqui podemos pedir que ele acrescente uma variável de contagem para cada elemento, retornando, ao final, uma variável de contagem com o número total de elementos. Vejamos um exemplo:

```
<contadorCertidaoPositiva> = 0,
foreach(<devedora> IN |devedoras|) {
    if (<devedor.certidaoTrabalhista> == "Positiva") {
        <contadorCertidaoPositiva> = <contadorCeritdaoPositiva> + 1
    }
}
```

Vejamos o que isso quer dizer. Em primeiro lugar, atribuímos zero ao contador. Isso porque inicialmente a quantidade é zero. Essa atribuição é necessária para "inicializar a variável". Essa inicialização é necessária pois existe uma diferença entre ser vazia e ser igual a zero. Quando a variável é vazia o sistema suspende a execução de qualquer código que utilize a variável, até que ela seja respondida pelo usuário ou preenchida por atribuição. Logo, estamos preenchendo ela por atribuição, já que ela é puramente interna ao código e não será exibida ao usuário.

Seguimos estabelecendo que o sistema deve passar por cada devedora do conjunto de devedoras e, caso a devedora tenha certidão positiva, some um na variável de contagem. Para deixar essa lógica mais clara, imagine que há quatro devedoras no vetor. A primeira delas tem certidão positiva, a segunda certidão negativa, a terceira certidão positiva e quarta positiva com efeito de negativa. Quando o sistema entra na execução do foreach ele olha para a primeira devedora e executa o código para ela. Ele avalia para ela a condição. Ela tem certidão positiva. Então o contador recebe como valor seu valor original, que era zero, mais um. O sistema segue, então, para o próximo elemento. Ele toma a segunda devedora e executa o código. A condição impõe que o sistema faça a soma apenas se ela tiver certidão positiva, mas ela tem certidão negativa, então o sistema não faz nada. O contador permanece igual a um. O sistema segue para a terceira devedora. Ela tem certidão positiva, então o sistema executa o que está dentro do if e faz com que a variável passe a valer seu valor original, que naquele momento é um, acrescido de um, totalizando, agora, dois. O sistema segue então para a quarta e última devedora. Ele vê que sua certidão não é positiva, mas sim positiva com efeito de negativa, então ele não executa o código encapsulado e a variável se mantém com o valor dois. Como todos os elementos foram avaliados a iteração se encerra e o sistema segue para executar as próximas linhas de código.

Como vimos por essa descrição específica do que está se passando por trás da tela vemos como a variável de contagem encerra a iteração com 2, justamente o número de empresas com certidão positiva.

Uma forma alternativa de proceder com a contagem é mediante ao uso de filtros. O seguinte algoritmo apresenta uma proposta alternativa ao algoritmo de contagem anterior:

```
1 |devedorasPositivadas| = |devedoras|.filter (
2     [this.certidaoTrabalhista] == "Positiva"
3 ),
4 <contadorCertidaoPositiva> = |devedorasPositivadas|.size()
```

O filtro é uma ferramenta que seleciona objetos dentro de um vetor do tipo Struct que respeitem determinado critério. Na Linha 2 do algoritmo anterior, esse critério é apontado a partir do identificador 'this', que significa a própria posição em iterada.

Foreach

O foreach é a forma mais comum de trabalharmos com vetores. Ele impõe que determinadas linhas de códigos sejam executadas para cada um dos elementos de um vetor. Como dito anteriormente, tal procedimento é chamado de iteração. Seu uso é frequente com partes, pois há um número indefinido delas e, quantas quer que elas sejam, o sistema deverá qualificar cada uma delas. Vejamos um exemplo:

```
1 print "\pAs Partes declaram que o disposto nesse Contrato não abrange as seguintes
2        sociedades: ",
3 foreach(<empresa> IN |excluidas|) where (separator = "%f1; %s1; %p1 e %l1.") {
4     print <empresa.nome> & ", " & <empresa.tipoSocietario>.lowercase() &
5           " inscrita no CNPJ sob o n° " & <empresa.cnpj>
6 },
7 print "\n\b"
```

Nesse exemplo iniciamos abrindo uma cláusula que exclui determinadas pessoas jurídicas do grupo do disposto no contrato. Após a introdução, pedimos que o sistema descreva quais são essas pessoas. Veja que informamos apenas uma vez o que queremos como descrição: O nome, o tipo de societário e o CNPJ. O que garante que esse procedimento será repetido é o foreach. Lendo o código em português na ordem estamos dizendo "para cada uma das empresas no conjunto de empresas do grupo excluídas do negócio, onde a forma de separar um elemento do outro é ponto e vírgula, imprima seu nome, o tipo societário e o CNPJ".

O "lowercase()" é um método aplicável a Strings. Ele tem por função colocar em letras minúsculas todo o texto. Ele foi invocado pois o conteúdo apresentado para o usuário em listagens geralmente vem com a primeira letra maiúscula. Assim, esse método obriga que esse conteúdo seja em letra minúscula no texto impresso.

O separador pode ser alterado. Conforme introduzimos anteriormente, o "f" corresponde a "*first*", o "s" a "*second*" e assim por diante. Nele definimos como separar os textos impressos, já que eles serão impressos várias vezes. No exemplo escolhemos separar os elementos entre si com ponto e vírgula, separando apenas o penúltimo do último por "e", como se costuma fazer na linguagem escrita. Por fim, após o último elemento imprimimos um ponto final.

Após fazer essas impressões pedimos que o sistema encerre a cláusula e pule uma linha. Um resultado possível dessa linha de código poderia ser o seguinte:

3. As Partes declaram que o disposto nesse Contrato não abrange as seguintes sociedades: Nintendo do Brasil, sociedade por ações inscrita no CNPJ sob o nº 12.345.678/0000-00; Atari do Brasil, sociedade limitada inscrita no CNPJ sob o nº 23.456.789/0000-00 e Sega do Brasil, sociedade em comandita simples inscrita no CNPJ sob o nº 55.555.555/8888-88.

Operadores lógicos

Frequentemente temos que criar condições complexas. Já vimos algumas em exemplos anteriores. Podemos buscar pessoas naturais com mais de 18 anos, pessoas naturais casadas, pessoas jurídicas com capital aberto, capital superior a 100 milhões de reais, mas não listadas no Novo Mercado. Para construirmos essas condições precisamos de operadores lógicos (os conectivos) como "**OR**", "**XOR**", "**AND**" e "**NOT**".

O operador OR tem um papel de união de conjuntos. O operador XOR é uma operação de OU exclusivo, não considerando a ocorrência mútua. Quando colocamos uma condição ou outra estamos dizendo que nosso conjunto de interesse é a união do conjunto do que satisfaz a primeira condição com o conjunto que satisfaz a segunda. Assim, ao colocar duas condições com OR entre elas estamos dizendo ao sistema para executar o trecho de código se ocorrer uma, se ocorrer a outra ou se ocorrerem as duas ao mesmo tempo.

Já AND desempenha o papel de união. Colocando num if suas condições com um AND entre elas implica que o sistema só executará o código se a primeira e a segunda forem verdadeiras. Por fim, o NOT desempenha o papel de conjunto complementar, ou seja, o sistema só executa o trecho do código se acontecer qualquer outra coisa que não a condição definida.

Diferentemente dos operadores anteriores, o NOT não é um operador binário, a saber, utilizado com duas condições necessariamente. Ele é dito ser operador unário, pois nega o termo imediatamente posterior a ele.

Talvez mais complexo do que o uso dos operadores é saber separá-los com parênteses. Se não delimitarmos bem o que queremos dizer podemos cometer erros de lógica graves. Suponha que queiramos, assim como no exemplo acima, inserir uma representação relativa a autorização de cônjuges. Queremos que ela seja impressa quando alguém é casado, ou vive em união estável, com qualquer regime que não seja o de separação total de bens. Vejamos:

```
1 if (EXISTS <devedor> IN |devedores| :
2            <devedor.tipoDeSujeito> == "Pessoa natural" AND
3            ( <devedor.estadoCivil> == "Casado" OR
4              <devedor.estadoCivil> == "Convivente estável") AND NOT
5            ( <devedor.regimeDeBens> == "Separação total de bens" )
6    )
```

Nessa ordem garantimos que (1) o devedor é pessoa natural, (2) que ele é casado ou convivente estável e (3) que seu regime de bens não é o de separação total de bens. Se não colocássemos parênteses, ficaria assim:

```
if (EXISTS <devedor> IN |devedores| :
            <devedor.tipoDeSujeito> == "Pessoa natural" AND
            <devedor.estadoCivil> == "Casado" OR
            <devedor.estadoCivil> == "Convivente estável" AND
            <devedor.regimeDeBens> ~= "Separação total de bens"
    )
```

Aqui temos um problema. Impomos que (1) a pessoa seja natural e casada ou (2) que ela seja convivente estável e não tenha o regime de separação total de bens.

Em primeiro lugar, isso retornaria um erro, porque só pessoas naturais podem ser conviventes estáveis, o que não é garantido na condição alternativa posterior ao OR.

Em segundo lugar, ainda que não houvesse esse erro de renderização, ou seja, o uso de uma informação que não sabemos se existe, como a avaliação do regime de bens está apenas na condição alternativa o sistema aceitaria situações em que a pessoa é apenas natural e casada, ainda que seu regime de bens seja o de separação total de bens. Isso porque se o sistema verifica que a primeira condição é verdadeira ele nem avalia a segunda, ele executa diretamente o código circunscrito pelo if. Nesse caso é o que ele faria após verificar que a pessoa é natural e casada. Essa falta de parênteses está, portanto, errada.

Ademais, tenha sempre em mete a ordem de precedência desses operadores: uso de parênteses sobre NOT, NOT sobre AND, e AND sobre OR (ou XOR). Lembre-se que na lógica Boolena, AND equivale a multiplicação/divisão enquanto que OR (ou XOR) equivalem a adição/subtração.

Por fim, tente sempre otimizar a forma com que você redige condições. É muito mais fácil dizer que você quer qualquer regime de bens que não o de separação total do que dizer que você quer o de comunhão total ou o de comunhão parcial ou o de participação final nos aquestos. Essas técnicas são análogas às que usamos para resolver exercícios de probabilidade básica na escola. Às vezes é muito mais fácil calcular a probabilidade dos demais cenários do que calcular a probabilidade de ocorrer a situação de interesse. Aqui é a mesma coisa, as vezes é mais fácil definir o conjunto e não nos interessa e negá-lo do que defini-lo diretamente.

Operações dentro de structs

Em Lawtex existe a possibilidade de usar operações dentro de Structs com o intuito de definir condições para existência de alguns fields internos, ou mesmo executar operações no momento em que determinados campos são respondidos.

Ifstruct: O ifstruct, como o nome sugere, é uma regra de existência condicional para campos da estrutura: Perguntar CPF apenas se temos uma pessoa é física, perguntar tipo societário apenas se temos uma pessoa jurídica, perguntar regime de bens apenas se temos uma pessoa física e casada ou em união estável e assim por diante. Impomos, por meio dessa sintaxe, uma condição para a existência de uma informação que caracteriza o objeto.

Lembre-se que essa operação é definida em abstrato, ou seja, ocorrerá na exibição de qualquer objeto concreto daquele tipo. Daí as operações serem feitas com notação de campo. Isso vale para qualquer operação dentro de estrutura. Vejamos um exemplo:

```
+[tipoDeSujeito] : List ("Pessoa natural", "Pessoa jurídica") {
    name = "Tipo de sujeito"
    atomic = true
    default = "Pessoa jurídica"
},
if ([tipoDeSujeito] == "Pessoa natural") {
    +[documento] : String where ("CPF") {
        name = "CPF"
    }
} else {
    +[documento] : String where ("CNPJ") {
        name = "CNPJ"
    }
}
```

Esse exemplo apresenta um caso especial de ifstruct, conhecido como **polimorfismo**. Ao considerar o campo [tipoDeSujeito] pessoa natural ou pessoa jurídica, o campo [documento] sempre existirá, contudo, com valores semânticos diferentes.

Embora em orientação a objetos, o polimorfismo tenha um significado diferente, nesse caso, é a propriedade de tratar vários campos protegidos por ifstruct de maneira homogênea, ou seja, aplicamos o mesmo nome ao campo em condições alternativas e definimos um default para que ele sempre exista. Em outras palavras, no início da execução, em virtude do default o campo de tipo de sujeito não é vazio, mas sim pessoa jurídica, então o número perante a receita existe, na modalidade CNPJ. Se o usuário modificar a resposta para qualquer outra o campo continuará existindo, ainda que com outro name, de forma que não precisamos mais garantir sua existência na parte operativa.

Observe que o ifstruct está mais relacionado com a definição dos campos do que com a operação deles por si. Isso porque a existência de um campo é condicionada. Sendo assim, deve-se tomar cuidado ao operar um campo "coberto" por um ifstruct. Nesses casos, deve-se sempre garantir que a condição de existência do referido campo coberto se cumpra no momento de sua operação (comparação, impressão, atribuição, iteração, etc.)

Erros de renderização: Como já adiantamos, os erros de renderização ocorrem quando utilizamos uma informação sem garantir sua existência. É importante emitir erros dessa natureza a fim de garantir a consistência do documento jurídica. Por exemplo, não se admite na lei Brasileira que uma pessoa jurídica sem nacionalidade brasileira possua RG. Nesse caso, é inconsistente pedir para um estrangeiro tal informação. Logo, por não requisitá-lo, maior erro ainda seria imprimir o RG de um estrangeiro "por acidente". Isso dificultaria muito a revisão humana do documento final produzido. Sendo assim, encare as mensagens de erro com bons olhos.

Qualquer uso não protegido vai gerar um erro. Considere o exemplo anterior re-escrito sem polimorfismo:

```
+[tipoDeSujeito] : List ("Pessoa natural", "Pessoa jurídica") {
    name = "Tipo de sujeito"
    atomic = true
},
if ([tipoDeSujeito] == "Pessoa natural") {
    +[cpf] : String where ("CPF") {
        name = "CPF"
    }
} else {
    +[cnpj] : String where ("CNPJ") {
        name = "CNPJ"
    }
}
```

Quando mandamos o sistema imprimir o CPF de um sujeito, sem garantirmos que esse sujeito seja pessoa física, o sistema acusará um erro. Assim, uma regra de bolso para prevenir esse erro é repetir a estrutura de ifs da estrutura nas operações. Se você impôs duas condições para que a informação seja perguntada na estrutura, imponha as duas mesmas condições para imprimí-la, multiplicá-la, concatená-la, atribuí-la para uma variável etc.

Conforme discutido anteriormente, o polimorfismo garante que um campo sempre exista. Para isso ele deve, além de existir um if (opcionalmente, com mais elseif), sempre deve existir um else, para que nenhum caso fique descoberto. Ademais, o papel do default é fundamental, como será visto adiante. No exemplo a seguir ocorre esse erro quando se tenta imprimir [documento] diretamente:

```
+[tipoDeSujeito] : List ("Pessoa natural", "Pessoa jurídica") {
    name = "Tipo de sujeito"
    atomic = true
},
if ([tipoDeSujeito] == "Pessoa natural") {
    +[documento] : String where ("CPF") {
        name = "CPF"
    }
} else {
    +[documento] : String where ("CNPJ") {
        name = "CNPJ"
    }
}
```

Embora se pareça muito com o polimorfismo, não é de fato. Isso porque [tipoDeSujeito] não foi **inicializado** por um default. Logo, quando o programa for executar a impressão do documento, por não haver default, o sistema não sabe desambiguar de onde pegar a informação, se do primeiro ou do segundo ifstruct. Para evitar que o campo coberto exista já no início da execução, os campos condicionais necessários não podem ser vazios: insira default nesses campos.

Exercício 54. Vamos lá, considere o exemplo anterior, crie uma Struct e mova o conteúdo do exemplo anterior dentro da propriedade fields. Adicione "Ente não personificado" à lista [tipoDeSujeito], e garanta que não hajam erros de renderização nas seguintes operações:

- Imprimir CPF e CNPJ.

Loaders: O loaders é a parte operativa da estrutura, comportando todas as formas de operações já descritas. Seu objetivo, contudo, é primordialmente fazer operações apenas com os elementos da própria estrutura para modificar a exibição. Vejamos um exemplo:

```
*struct[GarantiaContrato] {
   fields {
      +[tipoDeContrato] : String {
          name = "Tipo de contrato"
      },
      +[tipoDeGarantia] : List ("Real","Outras") {
          name = "Tipos de garantia"
          atomic = true
          default = "Real"
      },
      if ([tipoDeGarantia] == "Real") {
          +[garantiaReal] : List ("Hipoteca", "Penhor", "Alienação fiduciária",
                                  "Cessão fiduciária", "Penhora", "Outra") {
              name = "Tipos de garantia real"
              atomic = true
          }
      } else {
          +[garantiaFidejussoria] : List ("Fiança", "Caução", "Seguro fiança",
                                          "Outra") {
              name = "Tipo de garantia"
              atomic = true
          }
      }
   }
   loaders {
      if ([tipoDeContrato] == "Contrato de Locação") {
         [garantiaReal].options = "Cessão fiduciária",
         [garantiaFidejussoria].options = {"Fiança","Caução","Seguro fiança"}
      } elseif ([tipoDeContrato] == "Acordo judicial") {
         [tipoDeGarantia] = "Real",
         [garantiaReal].options = {"Hipoteca","Penhor",
                                   "Alienação fiduciária",
                                   "Penhora"}
      }
   }
}
```

A sistemática dessa estrutura é simples. O usuário insere o tipo de contrato, escolhe um tipo de garantia e lhe é apresentada uma lista de garantias, a depender do tipo escolhido. Ocorre que, como já sugerimos, nem toda garantia é pertinente a todo contrato. Logo, vemos no loaders que se o tipo de contrato inserido for de locação a única garantia real possível é a cessão fiduciária e, dentre as demais garantias, as únicas possíveis são a fiança, a caução e o seguro fiança. Por outro lado, no caso de acordo, a estrutura impõe que a única garantia possível seja a real, pois executa uma atribuição bloqueando a escolha do usuário. Por fim, em todas as demais opções a estrutura exclui da escolha do usuário a constituição de penhora, que só é possível em acordo judicial.

É frequente utilizarmos o loaders para modificar dinamicamente os elementos de uma lista de uma maneira mais enxuta do que com diversos ifstructs.

Ordenação de Cards

Visto que sabemos declarar operandos, é essencial sua operação. Um operando declarado mas não operado para nada serve. A operação é um aspecto importante da programação. É onde todas as informações são processadas, o raciocínio e o desenvolvimento da argumentação lógica é colocada em prática.

Como citado anteriormente, cada operando se torna um *card*. Esses *cards* são respondido pelo usuário do template a fim de dar substrato lógico para que o programa realize suas impressões. Os *cards* podem ser levados ao usuário na ordem em que a informação é usada ou em uma ordem determinada pela engenheira ou engenheiro jurídico.

Em geral, o uso da informação segue a lógica jurídica do template, que não necessariamente é a mais fácil do ponto de vista do usuário. Para isso, podemos usar uma funcionalidade que surgiu em exemplos acima, o método ask. Inserindo, por exemplo, a linha **|devedores|.ask()** forçamos o sistema a perguntar naquele momento aquelas informações.

É possível, dessa forma, ordenarmos as perguntas como quisermos no início da parte operativa do código e depois operar como desejamos. É possível, inclusive, modularizar a lógica que ordena as perguntas com um branch que chamamos de TOC, ou seja, *Table of Contents*.

Suponha que os *cards* de um template se dividem em (1) relativos às partes, (2) relativos ao objeto, (3) relativos ao preço e ao pagamento, (4) relativos às garantias, (5) relativos às assinaturas e testemunhas. Podemos criar um branch para cada um desses grupos, configurando index como true, e nas operações desses branchs apenas fazer o ask() dos operandos que queremos agrupar. Configurar o index como verdadeiro implica que queremos que aquele branch sirva para agrupar perguntas.

Por fim, podemos nomear cada um desses grupos de forma a identificar melhor cada grupo de perguntas. Ainda não discutimos o que são os branchs, mas por enquanto basta saber que podemos declará-los do jeito descrito e depois utilizá-los simplesmente para ordenar as perguntas.

Vejamos, por exemplo, como ficaria o branch de ordenação do grupo (1):

```
declarations {
    branch[TOC_Partes] {
        name = "Partes do contrato"
        operations {
            |devedoras|.ask(),
            |credoras|.ask(),
            |garantidoras|.ask(),
            |intervenientesAnuentes|.ask()
        }
    }
}
operations {
    use branch[TOC_Partes]
}
```

Exercício 55. Agora tente você. Como ficaria o branch de ordenação do grupo (5)?

Tubes

Dentre as operações, se destacam os **tubes**, que são funções ou métodos assinados em Lawtex, mas implementados em Java. O nome "tube" é um sinônimo de "*pipe*" (do inglês) que significa tubo: um canal de transferência de fluxos usado por diversos ramos da engenharia. Levando para o caso do Lawtex, esse "tubo" pode ser visto como um mecanismo usado para transferir fluxos de operações (trechos de código) para o núcleo do sistema Looplex. Assim, o programador Lawtex delega a responsabilidade da execução de um determinado trecho de código aos programadores de desenvolvimento da plataforma Looplex. Por exemplo, um programador Lawtex deseja enviar um e-mail dentro do sistema. É de se esperar que ele não conheça todos os protocolos de transferência de e-mail (tais como STMP, IMAP), bem como detalhes técnicos para sua implementação (portas e serviços). Assim, a equipe de desenvolvimento Looplex se responsabiliza por encapsular todos os comandos suficientes para enviar um email dentro de um método, e a partir disso, provê um ponto de acesso ao programador Lawtex para que seja possível realizar essa tarefa: *sendMail("lista de e-mails´´, "mensagem")*.

Uma função tem a sintaxe bem particular:

nomeDaFuncao(<arg1>, "arg2", etc()),

a qual inicia com letra minúscula e recebe zero, ou mais argumentos, sejam eles constantes, operandos, ou mesmo outras funções. Um método corresponde a uma função específica associada a uma classe semântica específica dentro do Lawtex. Um método sempre vem separado por ponto '.' de seu proprietário chamador (invoker):

<invoker>.nomeDoMetodo(<arg1>, "arg2", etc()).

Por fins didáticos, atualmente os tubes estão categorizados em oito categorias:

1. Tubes de texto;

2. Tubes de funções matemáticas;
3. Tubes coleções e acessores;
4. Tubes de data e horas;
5. Tubes de HTML e gráficos;
6. Tubes de API e depuração;
7. Tubes emuladores de comandos; e
8. Tubes emuladores e controladores de operandos.

São muitos os tubes dentro de cada classe! Ademais, tubes surgem todos os dias... Sendo assim, tomaremos apenas algumas classes, bem como, alguns tubes dessas classes como exemplo, apenas para exemplificar o uso dos mesmos. Para acompanhar atualizações e releases, consulte regularmente o link `http://dev.looplex.com/dokuwiki/doku.php` para obter uma documentação mais completa.

Tubes de texto: Os tubes de texto são ferramentas para formatação do texto. Não há muito o que explicar no seu uso, com exceção da gramática, então só listaremos as sintaxes e a função de cada um, quando a complexidade for baixa. Observe, contudo, que eles podem ser aplicados sobre qualquer String, seja ela diretamente escrita entre aspas, ou seja ela uma variável do tipo lista, variável do tipo lista atômica, campo de uma posição de vetor etc. Logo, nos exemplos abaixo, onde você ler <**texto**> saiba que o mesmo valeria para "texto" ou <objeto.campo> ou |vetor{2}.campo.subcampo|. Saiba também que é possível utilizar um tube dentro do outro, como quando queremos algo em negrito e em itálico.

Para colocar um texto em negrito basta inserir **bold(**<**texto**>**)** e para colocar um texto em itálico basta inserir: **it(**<**texto**>**)**. Para que o texto fique com recuo lateral de citação, basta digitar o comando **citation(**<**texto**>**)**.

Exercício 56. Teste seu conhecimento:

a) Faça a devida citação referente ao artigo 1º, abaixo indicado:

```
print "\pComo destacado, o valor da dívida foi atualizado monetariamente,
       acrescidos de juros de 1% (um por cento) ao mês. Assim dispõe a Lei
       Federal n.º 6.899/81:\b\b",
print "Art. 1º. A correção monetária incide sobre qualquer débito resultante
       de decisão judicial, inclusive sobre custas e honorários advocatícios.
       §1º Nas exceções de títulos de dívida líquida e certa, a correção será
       calculada a contar do respectivo vencimento.",
```

b) Coloque em negrito o título da cláusula abaixo:

```
print "Cláusula 7ª - Reconhecimento da Dívida",
print "\pA Dívida é reconhecida pelas Partes como líquida, certa e exigível,
       razão pela qual renunciam ao direito presente ou futuro de questioná-la,
       revê-la, investigar sua origem e resultado, tendo-a, pois como definitiva,
       expressando o valor de sua obrigação de pagar. A presente renúncia e
       desistência abrange não só a Parte Credora como também o Credor Original.
       \n\b\b"
```

c) Coloque em itálico e negrito a seguinte frase:

> A correção monetária - nada importa a natureza do crédito - deve incidir a partir do momento em que o devedor incidiu em mora. De outro modo, o inadimplente será beneficiado por enriquecimento ilícito. (STJ. 1ª Turma, REsp 11.882-0-SP, rel. Min. Humberto Gomes de Barros, j. 8.3.93, negaram provimento, v.u., DJU 26.4.93, p. 7.168, 2ª col., em.).

Aqui retomamos a gramática já descrita no item de print. Conforme dito naquele contexto, usamos o método **<sujeito>.grammar(<texto>)** para vincular uma palavra a um sujeito ou vetor de sujeitos de forma que o sistema fará a concordância de gênero e número da palavra com base na referência indicada. A concordância é feita pois há um dicionário no banco de dados que o sistema acessa para singular masculino, singular feminino, plural masculino e plural feminino. Para que seja conjugada a concordância nominal, a única restrição feita é que o tipo do operando que invocou o grammar tenha um campo especial declarado: o campo [genero], cujos valores devem ser "Masculino" ou "Feminino".

```
print "§ 4.º A reunião torna-se dispensável quando " & |socios|.grammar("o sócio")  &
      " " &|socios|.grammar("decidir")  &  ", por escrito, sobre a matéria que seria
      objeto dela.\b\b"
```

Para colocarmos todas as letras de um texto em caixa-alta, é suficiente aplicar a função **uppercase(<texto>)**. A função uppercase também é tida como método de variáveis do tipo texto, de modo que é válido escrever **<texto>.uppercase()**. O mesmo vale para as demais funções análogas. O tube lowercase deixa todas as letras minúsculas, o **firstUpper** torna a primeira letra do texto maiúscula e o **firstLower** a primeira letra do texto minúscula.

Podemos mudar o alinhamento de textos, colocando à direita, à esquerda ou centralizados. Isso é útil para posicionar nomes e títulos de maneira diferente.

Para tanto basta fazer **align(<texto>, "center")**, trocando o "center" por "left" ou "right", para alinhar ao lado.

Exercício 57. Faça o seguinte exercício, coloque no nome de um contrato de maneira centralizada.

Tubes de funções matemáticas: Tubes de operações matemáticas implementam funções e métodos para realização de tarefas que envolvem cálculos e operações matemáticas que extrapolam o domínio dos

operadores aritméticos.

Obviamente, os tubes de funções matemáticas são aplicáveis apenas a tipos numéricos, a saber: Integer, Real ou Currency. Sejam escritas diretamente como valores numéricos (ex. 57 ou 3.23), sejam variáveis, campos de uma posição de vetor numérico, ou qualquer outro operando numérico, onde você ler <**numero**> nos exemplos abaixo, saiba que o mesmo representa qualquer operando ou valor numérico. Assim como em todos os outros tubes, é possível utilizar um tube dentro do outro, como quando nos casos apresentado anteriormente nos tubes de texto.

Por exemplo, podemos calcular o módulo de um número com o uso do tube **abs(**<**numero**>**)**. Encontrar o máximo de um vetor de números é tão trivial quanto **max(|numero|)**. Ou ainda, se pode usar a função **round(**<**numero**>**, 2)** para arredondar um número com 2 casas decimais .

É preciso estar atento quanto aos parâmetros dos tubes de funções matemáticas. Se algum argumento não for numérico, um erro aparecerá.

Exercício 58. Teste seu conhecimento:

- O tube **mean** funciona de modo análogo ao tube **max**. Sendo assim, declare um vetor de números e calcule sua média.
- Implemente um algoritmo que use os tubes **floor** e **ceil** aplicados a números reais, e diga o que cada um deles fazem. Seu funcionamento é semelhante ao tube **abs**.

Tubes de coleções e acessores: Os tubes acessores e os tubes de coleções servem para acessar elementos dentro de vetores e listas, adicionar elementos, remover elementos, setar elementos default em vetores, entre outras operações. Representa uma classe de métodos e funções especializados em operar vetores e listas.

Por um lado, nos casos de listas vale destacar um tube que toma uma lista associativa e a partir de uma chave, retorna o valor correspondente. Tal tube é o <**lista**>**.getByKey("Nome da chave")**. Considere o seguinte exemplo de uma lista associativa declarada a seguir:

```
1 declarations {
2     *list[MoedasPorPaises] {
3         name = "Lista associativa de países e suas moedas respectivas"
4         fields = {"País", "Nome da moeda", "Código", "Unidade monetária"}
5         options = ({"Brasil", "real", "BRL", "R$"},
6                    {"Argentina", "peso argentino", "ARS", "$"},
7                    {"Colômbia", "peso colombiano", "COP", "$"},
8                    {"Estados Unidos", "dólar americano", "USD", "US$"})
9         type = "String"
10    }
11 }
```

Ao declarar uma lista podemos criar referências e associações entre informações internas. Assim, se um usuário responder apenas qual o país de uma variável do tipo desta lista, digamos <paisDeConstituição>,

estamos capturando toda uma série de referências vinculadas a este país, como o nome de sua moeda, código de sua moeda e seu símbolo monetário. Para recuperar o nome da moeda, por exemplo, use o seguinte comando **<lista>.getByKey("Nome da moeda")**, e assim por diante.

Por outro lado, nos casos de vetores, podemos adicionar um elemento oculto a um vetor existe através do seguinte comando: **|vetor|.add(<elemento>)**. Podemos também obter o primeiro elemento do vetor através do comando **|vetor|.get(0)**, assim como podemos excluí-lo: **|vetor|.remove(0)**.

Para fins de conhecimento, o comando **|vetor|.get(0)** é equivalente ao comando de indexação direta do vetor **|vetor{0}|**. Note que ao excluir um elemento de uma posição x, todas as outras posições superiores são reposicionadas de modo a se conformar ao novo vetor. Se por exemplo, temos um vetor com duas posições 0 e 1, ao excluirmos a posição 0, a antiga posição 1 passa a ser acessada como a nova posição 0. O uso da propriedade default é restrita a operandos não vetorizáveis. Logo, os vetores não possuem default como propriedade. Mas podemos simular esse uso através do método **|vetor|.set(0, <elem>)**. Nesse caso, <elem> será tido como resposta default da posição 0 de |vetor|.

Exercício 59. Teste seu conhecimento: Implemente os casos citados anteriormente para (a) um vetor de String aplicado aos seus respectivos tubes acessores, e (b) para a lista associativa na recuperação do símbolo da moeda.

Tubes de data e horas: Os tubes de data e horas representam as classes de métodos e funções especializados em operar valores ou operandos do tipo data e horário. Considere sem perda de generalidade que **<dateTime>** nos exemplos abaixo representam valores ou operandos do tipo data e horário.

Por exemplo, podemos extrair o dia de uma data do seguinte modo: **day(<dateTime>)**. Assim, o comando **day("04/08/1983")** retorna 4. O mesmo funciona para extração de ano (year), mês (month), horas (hour), minutos (minute) e segundos (second). Como foi citado na seção de onde se conceituou o tipo Date e Time, o tube **today()** retorna o dia de hoje, bem como o tube **now()** retorna o dia e horário atual.

Um par de tubes bem úteis são os tubes **before** e **after**. Esses tubes são usados para determinar se uma determinada data ocorreu antes (ou depois) de outra. No caso de

<dateTime_1>.before(<dateTime_2>),

o comando verifica se <dateTime_1> ocorre antes de <dateTime_2>, assim como no caso de

<dateTime_1>.after(<dateTime_2>),

o comando verifica se <dateTime_1> ocorre depois de <dateTime_2>.

Modificar datas é fundamental em documentos jurídicos. Um comando muito útil para alteração de datas, mediante a adição de x dias a uma data **plusDays(<date>, x)**, onde x é um número inteiro. O mesmo funciona para subtração: **minusDays(<date>, x)**. Existem outras variações para Years, Months, Hours, Minutes e Seconds, todas prefixadas com "plus" ou "minus".

Exercício 60. Teste seu conhecimento:

a) Um comando interessante é a escrita de datas por extenso: **expandedDate(<date>)**. Implemente um código em Lawtex que imprima o dia de hoje por extenso.

b) Implemente um código em Lawtex que imprima o dia de hoje, ontem e amanhã.

Tubes de HTML: Os tubes de HTML fornecem ferramentas para criação de elementos HTML mais sofisticados, bem como inserções de anexos ao documento, imagens gráficas, tabelas, hiperlinks, entre outros.

Um exemplo desse tipo de tube é o tube de listagem de itens. A listagem é o meio de criar bullet points em Lawtex. Para fazer listagem é preciso, em primeiro lugar, informar que iniciou-se a listagem e, depois, que a listagem se encerrou. Essa informação é uma operação a parte. Vejamos um exemplo:

```
print "Diante das razões expostas, requer-se:",
beginList(),
  print "\i " & "A produção de todas as provas em Direito admitidas; ",
  print "\i " & "A citação de " & <reu.nome> & " ;",
  print "\i " & "A condenação de " & <reu.nome> & " ao pagamento de " &
              <indenizacao> & " reais a título de indenização pelos danos morais
              sofridos pela parte autora. ",
endList(),
print "Termos em que pede deferimento."
```

Vemos no exemplo que o **beginList()** inicia a listagem e o **endList()** a encerra, sendo que cada novo ponto na listagem é identificado por "\i".

As tabelas de HTML também são muito úteis na composição de documentos jurídicos. Antes de mostrarmos como gerar tabelas é preciso compreender o que elas são conceitualmente em Lawtex: Imagine que alguém queira fazer, em uma planilha, um cadastro de clientes de uma loja. Intuitivamente, essa pessoa faria uma tabela com uma coluna para nome, outra para telefone, outra para endereço, outra para produtos comprados etc. Cada pessoa seria uma linha no cadastro. Quando um cliente novo surgisse, o lojista acrescentaria mais uma linha, perguntando ao cliente a informação de cada uma das colunas. Como essa tabela comporta tantos clientes quanto necessário, já podemos antever que a forma de fazer algo análogo será com um vetor, mas há algo faltando. Todos os clientes da tabela hipotética têm as mesmas colunas, ou seja, são compostos pelo mesmo conjunto de informações. Se eles são compostos pelo mesmo conjunto de informações, podemos dizer que eles partilham uma mesma estrutura. Esse é, portanto, o cenário.

A tabela é um vetor de estruturas. As colunas da tabela são os campos da estrutura. Cada linha da tabela é cada observação, ou seja, cada posição do vetor preenchida pelo usuário. Cada nova posição por ele adicionada será traduzida em uma nova linha na tabela.

Como as tabelas não são dinâmicas todas as colunas devem sempre existir para todas as observações. Logo, não podemos fazer tabelas com estruturas que usam ifstruct. Se quisermos fazer uma tabela com informações de estruturas com if temos de declarar uma estrutura dedicada apenas à tabela e atribuir em cada posição dela a informação vinda da estrutura mais complexa exibida ao usuário.

Para criar uma tabela basta usar a seguinte sintaxe:

```
1 table(|titulosDeDivida|.orderBy("valorDaDivida"), "vertical")
```

Basta colocar entre os parênteses da tabela a tabela com o critério de ordenação e, separado por vírgula, o posicionamento vertical ou horizontal da tabela. Essa ordenação deve ser feita com base em um dos fields.

Tubes emuladores de comandos: Os tubes emuladores de comandos servem para dar maior praticidade à programação. Em geral, eles combinam dois comandos em um só. Os comandos referidos são: *print*, *if* e *foreach*.

No caso do **printIf**, cuja sintaxe é:

printIf(<condicao>, "Texto caso a condição seja verdadeira", "Texto caso a condição seja falsa"),

se combina print a if. O método printIf já foi explicado anteriormente nas seção de definição de Print.

O mesmo ocorre com o tube **filter**, visto que é a combinação entre um foreach e um if, isto é, percorre por todos os elementos de um vetor e retorna apenas aqueles que satisfaz uma certa condição. O método filter já foi explicado anteriormente nas seção de definição de Vector. Sua sintaxe é:

|vetor|.filter(<condicao>).

Finalmente, podemos combinar print com foreach através do tube **vect2str**, o qual toma um vetor e uma String correspondente ao separator especificado e imprime os elementos desse vetor devidamente formatados. Sua sintaxe é:

vect2str(|vetor|, "%f, %s, %p e %f").

Vale salientar que caso o vetor seja do tipo Struct, a propriedade **id** será impressa.

Assim, cobrimos todos os pares de combinação entre as operações até aqui apresentadas.

Tubes emuladores e controladores de operandos: Essa classe de tubes serve para manipular e emular operandos. Por exemplo, uma imagem é um tipo de operando que não existe na definição dos tipos de operandos em Lawtex. Mas ainda assim, é possível a inserção de imagens dentro do documento ou como anexo ao próprio template.

As imagens podem ser trechos de documentos que devem constar no template, por exemplo, provas que deverão ser inseridas ou mesmo o logo e nota de rodapé com informações do escritório/empresa que deverão constar no template. Abaixo serão indicadas as sintaxes que permitem a inserção de imagens no template.

As imagens inseridas pelo usuário do template deverão ser inseridas por meio da função **upload**. A função upload, exige que o usuário escolha um arquivo para ser posteriormente anexado ao documento por meio do comando attach. A sintaxe do upload contém quatro parâmetros que deverão ser abordados pelo engenheiro jurídico ou engenheira jurídica, caso contrário o sistema não reconhecerá o comando.

No primeiro parâmetro deverá indicar o nome que deverá ser perguntado ao usuário, como por exemplo

"Termo de Rescisão (TRCT)".

Já no segundo parâmetro deverá constar o nome do arquivo com ".png", único formato de imagem aceito pelo sistema. Em outras palavras, todos os arquivos de imagens que serão inseridos no sistema necessariamente deverão constar no formato "png"ou não irão funcionar, como por exemplo "ImagemTRCT.png". São aceitos os formatos “png”, “jpg”, “bmp” e “pdf”.

No terceiro parâmetro o engenheiro jurídico ou engenheira jurídica deverá inserir o request (conforme anteriormente explicado), ou seja, a pergunta ao usuário acerca do documento que será inserido, como por exemplo, "Faça o upload do Termo de Rescisão do Contrato de Trabalho (TRCT) no formato .PNG".

No quarto parâmetro, deverá ser inserido a forma como a mensagem deverá ser visualizada pelo usuário e a obrigatoriedade de inserir a imagem no template, conforme explicado acima na funcionalidade do mandatory. Esse parâmetro também comporta três opções: (1) visível e obrigatório, representada pela sintaxe ‘+’; (2) visível, mas opcional, representada pela sintaxe ‘&’ e, por fim; (3) invisível, representada pela sintaxe ‘−’. Como exemplo, podemos indicar ‘+’, em caso de visível e obrigatório o seu preenchimento. Neste caso específico, o template apenas será finalizado quando a imagem for inserida no sistema pelo usuário. Essa informação é importante no momento de pensar na estruturação do template, pois sempre contemplará a hipótese de que uma imagem necessariamente deverá ser inserida pelo usuário.

Abaixo, segue a sintaxe a ser utilizada no caso de upload, utilizando os exemplos trazidos ao longo das explicações dos parâmetros utilizados na funcionalidade:

```
1 if (<descricaoVerbasTrabalhistasOuTRCT> == "Juntar cópia do TRCT aos autos") {
2     upload("Termo de Rescisão (TRCT)", "ImagemTRCT.png", "Faça o upload do
3             Termo de Rescisão do Contrato de Trabalho (TRCT) no formato .PNG", "+")
4 }
```

A funcionalidade do upload pode ser comparada à declaração de uma estrutura, pois nela constam as características que compõe a estrutura, sendo também necessária a indicação de como as variáveis da estrutura serão utilizadas na impressão. Nesse mesmo sentido, a funcionalidade upload precisa de uma funcionalidade para indicar o local em que será inserida a imagem, o que será indicado pela função **attach**.

Assim como a sintaxe do upload possui parâmetros fixos, a sintaxe do attach também possui quatro parâmetros que poderão ser inseridos pelo engenheiro jurídico ou engenheira jurídica. No entanto, apenas o primeiro parâmetro é obrigatório e os demais facultativos. Alguns comandos são os mesmos informados no upload, por isso, muita atenção para não inserir informações divergentes e inabilitar o comando.

No primeiro parâmetro deverá ser indicado o nome do arquivo com ".png", conforme indicado no segundo parâmetro do upload. Seguindo a linha de exemplos, seria "ImagemTRCT.png".

No segundo parâmetro o engenheiro jurídico ou engenheira jurídica deverá inserir um rótulo que explica o arquivo, como por exemplo "Termo de Rescisão (TRCT)".

Já no terceiro parâmetro deverá ser indicada a largura da imagem em pixels, por exemplo "600". Já o quarto parâmetro indicará a altura da imagem em pixels, por exemplo "900". Ambos os parâmetros devem ser setados com cuidado, pois distorcerá a imagem de entrada a fim de conformar esses parâmetros.

Abaixo, segue a sintaxe a ser utilizada no caso do comando attach, utilizando os exemplos trazidos ao longo das explicações dos parâmetros utilizados acima:

```
1 if (<descricaoVerbasTrabalhistasOuTRCT> ==  "Juntar cópia do TRCT aos autos") {
2     print "no Termo de Rescisão do Contrato de Trabalho (TRCT) abaixo.\n\b\b",
3     attach("ImagemTRCT.png", "Termo de Rescisão do Contrato de Trabalho")
4 }
```

Não se esqueçam, o comando attach deverá ser inserido no local em que a respectiva imagem deverá aparecer no template.

É possível também subir uma imagem vinculada a um template e vinculá-la diretamente ao template. Para que isso seja possível, basta utilizar o attach de modo independente, tomando o cuidado para subir a imagem ao sistema com o mesmo nome definido no argumento do tube attach. Por exemplo, operamos um attach sem o upload correspondente:

```
1 attach("ImagemTRCT.png", "Termo de Rescisão do Contrato de Trabalho")
```

Logo após isso, podemos subir a imagem "ImagemTRCT.png" pelo sistema da Looplex. Apenas tome cuidado, pois o nome deve respeitar letras maiúsculas e minúsculas.

Exercício 61. Agora vamos passar aos exercícios para treinar a sintaxe da funcionalidade de imagens:

a) Informe a sintaxe para a inserção de uma imagem de uma fatura, que necessariamente deverá aparecer para o usuário.

b) Informe a sintaxe para a inserção da tela de cadastro de inadimplentes, de maneira facultativa.

Existem uma série de outras classes de tubes, dentre as quais descrevem tubes para envio de e-mail, montagem de glossários, referências cruzadas, impressão de números por extenso, comunicação com APIs externas, inserção de gráficos, entre outros, que podem ser consultados na documentação oficial (`http://dev.looplex.com/dokuwiki/doku.php`).

Nós e Modularização

Nós são agregações de lógicas, encapsulamento de códigos, modularizações. Essas modularizações permitem o uso reiterado de uma lógica em vários contextos e a organização do código. Os nós podem ser independentes, ou seja, simplesmente usados no corpo do código, ou podem funcionar como funções matemáticas, recebendo um input e devolvendo um output. Os nós podem ser interpretados como pequenos templates. Eles têm nome, descrição, declarações e operações próprias. Outra vantagem de modularizar o código é que podemos inserir tags para a lógica.

O Lawtex permite modularizar o código mediante a declaração de um nó na íntegra. São nós em Lawtex os Tópicos, Branches, Períodos, Dependências e Loops. Futuras referências a esses nós são feitas com o comando de uso, como por exemplo o uso de um tópico local previamente declarado:

use topic[TOP_RealidadeDosFatos].

Assim, se programarmos cada cláusula de um contrato como um tópico, além de podermos usá-las em outros contratos, podemos inserir em cada uma delas tags tais como referências a artigos, leis e precedentes a elas aplicáveis. Veremos cada tipo de nó a seguir.

Topic e Branch: Esses nós são praticamente idênticos. A única diferença entre eles é que o Topic, imprime um título associado a ele, de forma que sempre que ele for utilizado esse título será impresso e o que estiver dentro dele será numerado em relação a ele. Vejamos um exemplo de branch:

```
branch[BRC_DescricaoAutores] {
    name = "Descrição dos autores"
    operations {
        foreach(<reu> IN |reus|) where (separator = "%f2, %s2, %p2 e %l2") {
            print <reu.nome> & printIf(<reu.tipoDeSujeito> == "Pessoa natural",
                                ", com inscrição no CPF sob o nº " & <reu.cpf>,
                                ", com inscrição no CNPJ sob o nº " & <reu.cpf>)
            },
        if (|reus|.size() > 1) {
            print ", partes rés já qualificadas nos autos deste processo"
        } else {
            print ", parte ré já qualificada nos autos deste processo"
        }
    }
}
```

É o que usamos para cláusulas ou argumentos com títulos como “Do Dano Moral” ou “Representações e Garantias”. O ideal é que, tirando assinaturas, cabeçalhos, endereçamentos e considerandos, você nunca imprima nada fora de tópicos. Os textos devem estar sempre dentro de tópicos e sempre entre “\p” e “\n”, para que sejam numerados, como já destacamos.

Seguimos agora com um exemplo de tópico:

```
topic[TOP_ClausulaInterpretacao] {
    name = "Interpretação"
    title = "Interpretação"
    operations {
        print "\pOs títulos e cabeçalhos deste Contrato servem meramente para
               referência e não devem limitar ou afetar o significado atribuído à
               cláusula a que fazem referência. Os termos desse contrato deverão
               ser interpretados com base na prática do mercado para esse tipo de
               operação, nunca podendo ser interpretados de forma a restringir as ",
        if (|devedores|.size() > 1) {
            print "obrigações das Partes Devedoras"
        } else {
            print "obrigações da Parte Devedora"
        },
        print ".\n"
    }
}
```

Loop: Esse tipo de nó encapsula uma estrutura de repetição para usos posteriores. O Loop é basicamente um branch que só permite o uso de um foreach. Tal foreach é escrito dentro de sua propriedade operation (está no singular por admitir apenas um único comando). O nó loop foi criado a fim de modularizar o comando foreach. Sua sintaxe é similar aos outros nós apresentados:

```
*loop[LOOP_EscreveAutor] {
    name = "Loop para escrever nomes de autores"
    description = "Este loop escreve uma lista de nomes de autores"
    separator = "%f1, %s2, %p2 e %l2."
    tags { "Qualificação", "Autor" }
    declarations {
        +|autores| : Vector[*Autor] {
            name = "Nome do autor"
        }
    }
    operation {
        foreach (<autor> IN |autores|) {
            print <autor.nome>
        }
    }
}
```

Conforme se observa, destaca-se a propriedade separator (a mesma usada na cláusula where) para fins de separar elementos do vetor por vírgulas e outros elementos conectivos da língua portuguesa. Vale salientar que as tags descritas nesse código, servem para indexar buscas e estão disponíveis para todos os nós.

Dependency: Já a Dependência encapsula uma estrutura de decisão. Observe que o uso de nós ajudam a deixar o código mais organizado. Sua sintaxe é similar aos outros nós apresentados:

```
*dependency[DEP_QualificacaoLogradouro] {
    declarations {
        +<endereco> : *Endereco
    }
    operation {
        if (<endereco.pais> == "Brasil") {
            use branch[BRC_QualificacaoLogradouroBrasileiro]
        } else {
            use branch[BRC_QualificacaoLogradouroEstrangeiro]
        }
    }
}
```

Dependency é um branch que só permite o uso de um if dentro da propriedade operation (note que também é no singular). Esse exemplo foi propositalmente feito usando branches declarados anteriormente. O uso de um nó com nome intuitivo (nesse caso um branch) é de fundamental importância para compreensão macro do código. Modularização é sempre um bom caminho a se tomar. Essa dependência é global, devidamente sinalizada com asterisco.

Period: Por fim, o Period encapsula um comando de impressão: print. Observe por fim a declaração de um período:

```
period[PER_QualificacaoLogradouroBrasileiro] {
    description = "Período de qualificação de logradouro brasileiro."
    tags {"Logradouro", "Endereço", "Localização", "Brasileiro", "Brasileira"}
    declarations {
        +<endereco> : *Endereco
    }
    operation {
        print {
            [version = <endereco.principal> & ", " & <endereco.numero>
                & printIf (<endereco.complemento>.isNotEmpty(),
                  " " & <endereco.complemento> & ",", "") & <endereco.bairro>
                & " CEP nº~" & <endereco.codigoPostal> & ", na Cidade de "
                & <endereco.cidade> & ", " & <endereco.uf> & ", " & <endereco.pais>]
        }
    }
}
```

Esse período é local, pois não está sinalizado com asterisco. Além disso, ele possui apenas uma versão declarada. Múltiplas versões podem ser declaradas e setada com o tube printVersion(x), onde x é um número que indica a ordem da versão a ser impressa a partir daquele momento. Note que operation também está no singular, assim como em loops e dependências.

Se ao invés de compormos grandes aglomerados de operações e lógicas em branches complexos, criarmos um nó para cada conjunto de operações com um sentido comum, o código será lido com mais facilidade e o reaproveitamento também será facilitado. Para ver um exemplo de cada um desses nós, basta usá-los no snippet. Você não verá grandes diferenças com os demais.

Escopo: Considerar o escopo onde os operandos estão sendo declarados e operados é de fundamental importância para o gerenciamento do fluxos de informações de um documento. Saber aonde foram declaradas as variáveis e os objetos, além de ser essencial para estrutura do template, também evita erros inesperados.

É comum nos depararmos com muitos erros de declaração ocasionados por escopos inadequados de operandos. Esses erros ocorrem porque o sistema não encontra o operando respectivo desde o onde foi usado até o nó raiz na árvore que define o template. Nesse caso, o nó raiz do template é o metainfo.

Os nós locais costumam ser declarados no metainfo (ou outro nó) e utilizados no ponto em que se deseja seu texto, contanto que esteja hierarquicamente contextualizado. Já os nós globais podem ser declarados até mesmo fora de qualquer áreas de declaração (declarations).

Lembre-se que quando falamos da topologia do template dissemos que tudo que você for utilizar em todo o template deveria ser declarado no metainfo. Isso porque o que for declarado, por exemplo, no corpo do documento, não será visível no cabeçalho e vice-versa. Com os nós a lógica é a mesma. O que for declarado em um nó não será visível para o outro, a não ser que um nó seja interno ao outro.

O que delimita se um nó é interno ao outro ou não é o fluxo operativo, isto é, se um nós foi usado dentro de outro ou não. Lembrando que a sintaxe de uso de um nó é através do comando **use**.

Para nós globais, as informações utilizadas sempre virão do contexto em que forem usados. Dessa forma, essas informações terão a notação externs, de forma que o uso do componente em um template precisa informar quais informações do template correspondem às informações externas do nó global utilizado. Segue um exemplo bem simples de declaração e uso de nós.

Componentes globais e componentes locais: Como já destacamos, existem componentes globais, ou seja, há estruturas e nós que podem ser utilizados em templates. Isso acelera a produção de novos templates, pois componentes utilizados em mais de um documento podem ser simplesmente reaproveitados. Além disso se atualizamos o componente, todos os templates que utilizam o componente serão automaticamente atualizados.

Como esse material é introdutório não estudaremos com profundidade o funcionamento dos componentes globais, apenas descreveremos quais são suas funcionalidades para que você saiba pelo menos quais informações buscar nos próximos materiais. No entanto, segue um exemplo para que você entenda um pouco mais sobre escopos de nós globais e locais.

```
1  declarations {
2     +<stringDeFora> : String,
3      topic[TOP_Local] {
4          operations {
5              +<stringInterna> : String
6          }
7          declarations {
8              <stringDeFora> = "Esse texto será recuperado lá fora"
9              <stringInterna> = "Esse texto será perdido lá fora"
10         }
11     },
12     *topic[TOP_Global] {
13         operations {
14             +<stringExterna> : externs String,
15         }
16         declarations {
17             <stringExterna> = "Esse texto será recuperado lá fora"
18         }
19     }
20 }
21 operations {
22     use topic[TOP_Local],
23         use *topic[TOP_Global] where (<stringExterna> : <stringDeFora>)
24 }
```

O uso das variáveis dentro dos tópicos obedecem o seguinte escopo:

(a) Dentro do tópico TOP_Local, a variável <stringInterna> (Linha 5) tem visibilidade restrita ao tópico. Uma vantagem do tópico local é a visibilidade irrestrita de todos os operandos declarados ao longo de todos os nós da rota de execução. Assim, o tópico local consegue manipular livremente <stringDeFora>.

(b) Dentro do tópico TOP_Global por sua vez, a variável <stringExterna> (Linha 14) é apenas um "proxy" (um substituto) para o operando externo a ser acoplado dentro da diretiva where. Nesse caso, a <stringExterna> é apenas um operando que representa <stringDeFora>. Esse acoplamento garante que qualquer alteração em <stringExterna> feita dentro do tópico global, reflita integralmente em <stringDeFora>.

Para usar essas componentes declaradas (Lists e Structs), basta tratá-las como um tipo primitivo, ou seja, declarar os operandos de tal maneira:

```
+<objetoLocal> : StructLocal {
    name = "Nome do objeto local"
                request = "Informe os dados do objeto local"
},
+<objetoGlobal> : *StructGlobal {
    name = "Nome do objeto global"
                request = "Informe os dados do objeto global"
},
+<objetoLocal> : ListLocal {
    name = "Nome da lista local"
                request = "Escolha um elemento da lista local"
                atomic = true
},
+<objetoGlobal> : *StructGlobal {
    name = "Nome da lista global"
                request = "Escolha alguns elementos da lista global"
                atomic = false
}
```

Acesso a informações internas a nós e às raízes de estruturas: É possível acessar informações que estão fora do escopo ou permitir que uma estrutura acesse uma informação da estrutura que a contém, chamada de raiz. Nesses casos excepcionais você deve identificar em que parte do template ou em qual nó está a informação que você deseja. Da mesma forma, na estrutura é preciso informar que a informação desejada está na raiz.

```
if ("Aline Mendes" IN [Metainfo:convocacaoPresenca.membrosDoConselho] OR
    [Metainfo:convocacaoPresenca.temCemPorCento] == true) {
      +[comoVotouFC] : List ("Aprovou", "Rejeitou", "Se absteve") {
          name = "Voto da Sr. Aline Mendes"
          request = "Indique como a Sra. Aline Mendes deliberou
                     sobre a aprovação de contas"
          atomic = true
          default = "Aprovou"
      }
}
```

Padrões e técnicas são criados dia após dia de modo a melhor utilizar todas essas funcionalidades apresentadas nesse módulo. Fique atento a atualizações do link `http://dev.looplex.com/dokuwiki/doku.php`.

6 Conclusões

Como explorado nesse material, a programação para advogados é uma realidade tangível. Programar é uma arte, lembre-se disso. ;-) O programador não deve simplesmente vomitar código sem qualquer critério. É preciso haver disciplina e muita organização para gerenciar seu código. Há pessoas que estimam o código produzido como se fosse um filho. Isso mesmo, seu código é um filho seu. Ele nasce, e você precisa dar atenção total a ele! E quando cresce, precisa dar mais atenção. Mas tenha certeza, num futuro próximo ele te trará alegrias e recompensas pelo tempo investido.

Uma dica final: otimize seu *template*. Revisite-o com frequência, fazer ajustes e modificações para ajustá-lo é sinal de maturidade! Selecione perguntas mais simples e ordene-as da maneira mais intuitiva, de forma a absorver o máximo de informação do usuário sem exigir dele um esforço excessivo. Pode parecer complexo para quem ainda não colocou "profissionalmente" as mãos no código, mas com o tempo você verá que isso é relativamente fácil e que, com o tempo, qualquer um é capaz de identificar pontos que poderiam ser otimizados.

Imagine, por exemplo, que numa ação que discute um contrato queremos programar alegações de ilegitimidade ativa. Para tanto, perguntamos quem são as partes autoras, em seguida perguntamos quem são as partes do contrato, comparamos seus dados e, quem for autora, mas não for parte, será considerada parte ilegítima. Inicialmente, poderíamos até aceitar essa lógica, mas assim que testássemos veríamos que, sempre que as partes autoras fossem também partes no contrato, o usuário seria obrigado a preencher duas vezes a mesma informação e você teria que programar uma lógica complexa de comparação dos dois conjuntos. Muito mais fácil é, dentre os dados de cada parte autora, perguntar se a parte inserida é também parte no contrato. O usuário tem apenas que olhar o contrato e buscar a parte lá, respondendo sim ou não dependendo do que encontrar. A avaliação também é simples. Se a parte autora tiver um "não" como resposta na pergunta alegamos de imediato a ilegitimidade. Esse tipo de otimização não é tão complexa por duas razões. Em primeiro lugar, porque você mesmo perceberá, pela complexidade das suas operações, que talvez suas perguntas não estejam ajudando. Em segundo lugar, porque ao contrário do que ocorre em outras circunstâncias no mundo da computação, somos advogados programando questões jurídicas.

Nós sabemos as necessidades do usuário, de forma que é mais fácil programá-las. É claro que um advogado com ampla experiência na área penal tenha alguma dificuldade para antever as necessidades de um usuário especializado em Direito previdenciário e vice-versa. No geral, um conhecimento sólido dos fundamentos de cada área do Direito permite que a programação seja eficiente. Com teste e releitura do código você perceberá quando está fazendo perguntas redundantes. Um cuidado, contudo, que é necessário é na precisão das perguntas. Fazer um *template* com perguntas vagas como "é cabível indenização por danos morais?" ou com campos muito abertos como "insira o argumento que sustenta a alegação de danos morais" transforma o *template* num formulário, diminuindo o seu valor. Todavia é uma questão mais de disciplina e dedicação do que de habilidade ou experiência. Um *template* feito com cuidado e atenção dificilmente será pobre a ponto de se resumir a um formulário.

Outros tópicos mais avançados como padrões Lawtex para o desenvolvimento eficiente de *templates* estão no radar e no próximo material traremos ao leitor uma visão mais completa sobre programação. Tenha em mente do que está por vir, pois o mercado jurídico nunca mais será o mesmo.□

Referências Bibliográficas

[1] J. Goodman, *Robots in Law: How Artificial Intelligence is Transforming Legal Services.* ARK Group, 2016.

[2] H. Bakhshia, J. M. Downingb, M. A. Osborneb, and P. Schneiderd, "The future of skills: Employment in 2030," 2017.

[3] C. N. de Justiça CNJ BRASIL, "Justiça em números 2017: ano-base 2016," 2017.

[4] A. B. de Lawtechs e LegalTechs, "Radar," 2017.

[5] W. Bonsor, Harper, "The mindset of legal inovation," *Law and Robots*, pp. 11–16.

[6] R. Susskind, *Tomorrows Lawyers*, 2nd ed. Oxford University Press, 2017.

[7] A. Goodman, "A better way to deal with electronically stored information," in *American Bar Association. Section of Litigation and American Bar Association.*, vol. 43, no. 1. American Bar Association, 2016.

[8] R. E. Susskind, *The end of lawyers?: rethinking the nature of legal services.* Oxford University Press, 2010.

[9] R. Dworkin, *Law's empire.* Harvard University Press, 1986.

[10] S. R. Safavian and D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE transactions on systems, man, and cybernetics*, vol. 21, no. 3, pp. 660–674, 1991.

[11] M. Roach, "Toward a new language of legal drafting," *arXiv preprint arXiv:1507.05081*, 2015.

[12] K. Schwaber and M. Beedle, *Agile software development with Scrum.* Prentice Hall Upper Saddle River, 2002, vol. 1.

[13] M. Szabo, "Design thinking in legal practice management," *Design Management Review*, vol. 21, no. 3, pp. 44–46, 2010.

[14] T. Buzan and B. Buzan, *How to mind map.* Thorsons London, 2002.

[15] P. P.-S. Chen, "The entity-relationship model - toward a unified view of data," in *Readings in artificial intelligence and databases.* Elsevier, 1988, pp. 98–111.

[16] M. Vianna, *Design thinking: inovação em negócios.* Design Thinking, 2012.

[17] R. L. Keeney, "Value-focused brainstorming," *Decision Analysis*, vol. 9, no. 4, pp. 303–313, 2012.

[18] S. G. Isaksen, *A review of brainstorming research: Six critical issues for inquiry.* Creative Research Unit, Creative Problem Solving Group-Buffalo New York, 1998.

[19] V. R. Brown and P. B. Paulus, "Making group brainstorming more effective: Recommendations from an associative memory perspective," *Current Directions in Psychological Science*, vol. 11, no. 6, pp. 208–212, 2002.

[20] T. Buzan and B. Buzan, *Mind Map Book, 1/e.* Rajpal & Sons, 2010.

[21] G. Booch, J. Rumbaugh, and I. Jacobson, *UML: guia do usuário.* Elsevier Brasil, 2006.

[22] D. B. West *et al.*, *Introduction to graph theory.* Prentice hall Upper Saddle River, 2001, vol. 2.

[23] H. d. A. Camargo, "Teoria dos grafos," 2013.

[24] A. M. Coelho, "Teoria dos grafos," 2013.

[25] G. M. R. Pereira and M. A. da Câmara, "Algumas aplicações da teoria dos grafos," *FAMAT em Revista*, no. 11, 2008.

[26] E. Prestes, "Teoria dos grafos," *Porto Alegre. 2011a. Disponível em:< http://www. inf. ufrgs. br/˜ prestes/Courses/Graph% 20Theory/GrafosA6. pdf>. Acesso em*, vol. 4, 2016.

[27] M. Roach, "Toward a new language of legal drafting," *J. High Tech. L.*, vol. 17, p. 43, 2016.

[28] W. Vincent III, "Legal xml and standards for the legal industry," *SMUL Rev.*, vol. 53, p. 1395, 2000.

[29] D. Arnold, "Writing scientific papers in latex," *College of the Redwoods. http://online. redwoods. edu/instruct/darnold/linalg/latex/project_ latex. pdf*, 2001.

[30] L. Torvalds, J. Hamano, and A. Ericsson, "Git manual page," 2017.

[31] B. Castrucci, *Elementos de teoria dos conjuntos.* Grupo de Estudos do Ensino da Matemática, GEEM, 1969.

[32] P. G. Hinman, "Fundamentals of mathematical logic," *Bulletin of Symbolic Logic*, vol. 13, no. 3, pp. 363–365, 2007.

[33] M. J. Cresswell and G. E. Hughes, *A new introduction to modal logic.* Routledge, 2012.

[34] J. Souza, *Lógica para ciência da computação.* Elsevier Brasil, 2017.

[35] J. M. Abe, *Introdução à Lógica para a Ciência da Computação.* Arte & Ciência, 2002.

[36] A. Tucker and R. Noonan, *Linguagens de Programação-: Princípios e Paradigmas.* AMGH Editora, 2009.

[37] F. d. A. R. Junior, "Programação orientada a eventos no lado do servidor utilizando node. js," 2012.

[38] S. Finne and S. P. Jones, "Programming reactive systems in haskell," in *Functional Programming, Glasgow 1994.* Springer, 1995, pp. 50–65.

[39] E. K. Piveta *et al.*, "Um modelo de suporte a programação orientada a aspectos," *Dissertação de mestrado*, 2001.