Universidade Federal Fluminense

Instituto de Computação

Departamento de Ciência da Computação

Hugo Farias Silva

LEARN - UMA LINGUAGEM PARA DESCRIÇÃO DE CURSOS ONLINE

Niterói-RJ

2016

HUGO FARIAS SILVA

LEARN - UMA LINGUAGEM PARA DESCRIÇÃO DE CURSOS ONLINE

Trabalho submetido ao Curso de Bacharelado em Ciência da Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Christiano de Oliveira Braga

Niterói-RJ

2016

Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e Instituto de Computação da UFF

S586 Silva, Hugo Farias
LEARN : uma liguagem para descrição de cursos online / Hugo Farias Silva. – Niterói, RJ : [s.n.], 2016.
58 f.

Trabalho (Conclusão de Curso) – Departamento de Computação, Universidade Federal Fluminense, 2016.
Orientador: Christiano de Oliveira Braga.

1. Teoria dos autômatos. 2. Ensino à distância. I. Título.

CDD 005.73

HUGO FARIAS SILVA

LEARN - UMA LINGUAGEM PARA DESCRIÇÃO DE CURSOS ONLINE

Trabalho submetido ao Curso de Bacharelado em Ciência da Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Aprovado em março de 2016.

BANCA EXAMINADORA

Prof. CHRISTIANO DE OLIVEIRA BRAGA, D.Sc. - Orientador

UFF

Prof. BRUNO LOPES VIEIRA, D.Sc.

UFF

Profa. ALINE MARINS PAES CARVALHO, D.Sc.

UFF

Niterói-RJ

2016

Aos meus pais, que desde cedo me ensinaram o valor do estudo.

Agradecimentos

Agradeço primeiramente a Deus, princípio e fim de tudo e razão de estar aqui.

Também aos meus pais, Cássio e Aparecida, que me fizeram quem sou.

Ao meu orientador, Christiano, pela oportunidade e por tudo que pude aprender.

Ao professor Paulo Blauth, cujo trabalho me acompanhou desde o começo da graduação e foi um alicerce neste projeto e pelo apoio que deu para que pudéssemos construir o que foi feito. Junto a ele, ao Departamento de Informática Teórica da UFRGS, pelos recursos que permitiram maior dedicação.

E a todos aqueles que me ajudaram, de algum modo, a chegar até aqui.

Resumo

O uso de plataformas digitais para auxiliar a educação é uma realidade crescente, seja como estruturas auxiliares ou para fornecer cursos completos à distância. Contudo, pessoas diferentes assimilam o mesmo conteúdo de formas diferentes, sendo assim um mesmo curso pode ser estudado por *caminhos* de aprendizagem diferentes. Ainda, a diversidade de tecnologias gera a necessidade de um meio de especificação multiplataforma.

Como solução para ambos os problemas, sugerimos uma *linguagem* declarativa, Learn, que assuma por padrão a concorrência entre todos os elementos atômicos de um curso e possa ser exportada para os diferentes sistemas de gerência de aprendizado disponíveis.

Neste texto apresentamos Learn através de sua sintaxe e sua semântica formal e damos um protótipo de transformador das suas descrições para gramáticas regulares, teorias de reescrita e páginas web, formalizando a abordagem de múltiplos caminhos de aprendizagem e possibilitando a exportação multiplataforma.

Palavras-chave: *Ensino Eletrônico, Educação à Distância, Educação Assistida por Computador, Métodos Formais, Lógica de Reescrita, Teoria de Autômatos.*

Abstract

The use of digital platforms to support the education is a growing reality, either as auxiliary structures or to provide entire distance courses. However different people assimilate the same content in different ways, therefore the same course can be studied by different learning *paths*. Furthermore the diversity of technologies makes the necessity of a way of multiplatform specification.

As a solution to both the problems we suggest a declarative *language*, Learn, which assumes by default the concurrence of all atomic elements of a course and can be output to the different available learning management systems.

In this text we introduce Learn throw its syntax and its formal semantics and give a transformer prototype from its descriptions to regular grammars, rewriting theories and web pages, formalizing the approach of multiple learning paths and making possible the multiplatform output.

Keywords: *E-learning, Distance Education, Computer-aided Education, Formal Methods, Rewriting Logic, Automata Theory*

Sumário

Lista de Figuras

Capítulo 1

Introdução

As novas tecnologias para assistência digital à educação vem facilitando o apoio às aulas presenciais e à educação à distância. Uma pesquisa simples na Internet revela ambientes de ensino como edX [1], Coursera [2] e Udacity [3] e plataformas largamente utilizadas, por vezes em mais de uma instalação por instituição, os MOOCs (*Massive Open Online Courses*) como o Moodle [4] e o TelEduc [5]. Estes se destacam como ferramentas para o gerenciamento do processo de aprendizado.

Ainda assim, a preparação de um curso num MOOC não é um processo trivial e cada um deles tem suas próprias particularidades e idiossincrasias tecnológicas. Isto motivou trabalhos como SCORM e IMS Learning Design [6], que objetivam criar uma interface única para elaboração de cursos online que permita sua instanciação em diversas tecnologias diferentes.

Há ainda uma limitação na abordagem usada pelos cursos online. É frequente a imposição de uma sequência rígida e inalterável dos assuntos a serem ministrados, o que, para a maioria dos casos, não tem justificativa aparente num contexto de aprendizado autodidata ou assistido. Dificilmente tal modelo poderá atender a todos, dado que diferentes pessoas apresentam diferentes ritmos de aprendizado. Em oposição, é também comum encontrar casos onde as unidades que compõem o curso são apresentadas simultaneamente ao aluno. Aí há uma evolução, pois é possível adequar a ordem de acordo com a necessidade. Contudo, há casos onde dois ou mais conteúdos possuem uma genuína relação de dependência. À aplicação de tais dependências sobre um contexto de liberdade das unidades chamamos *estratégia de ensino*. Sendo assim vemos o estudo como uma concorrência das unidades cujos requisitos já foram atendidos, seguindo o caminho que mais convier ao aluno.

Por exemplo, em um curso qualquer um estudante pode começar pela motivação do es-

tudo ou ir direto para as definições, seja começando pelas mais fundamentais ou pelas finais para só então buscar as mais básicas de acordo com a necessidade. Ou ainda pode começar pelos exemplos para só então consultar a explicação. Contudo, em sua estratégia o professor pode restringir que os exercícios só sejam acessíveis depois de estudada a matéria a qual se referem.

Para solucionar ambos os problemas levantados propomos uma única solução: uma linguagem declarativa, Learn, que tenha o poder de descrever cursos de forma abstrata, apontando seu conteúdo e detalhando sua estratégia. A razão para tal é que linguagens são ferramentas simples e podem ser usadas para a elaboração do curso sobre um modelo teórico, do qual seriam geradas representações em diferentes formatos que podem ser inseridos em plataformas de ensino como Moodle. Este processo focaria a preparação do curso unicamente em seus conteúdos e nas relações entre eles, não nas questões técnicas da plataforma a ser utilizada; e ainda permite que a mesma especificação possa ser reaproveitada em outras instanciações.

Learn é pensada para ter uma sintaxe simples e uma semântica formalmente definida capaz de conter os conteúdos de um curso divididos em unidades, os objetos de aprendizagem (vide o Apêndice A), com estratégias acopladas que determinam dependências entre esses objetos. Em um curso sem estratégias, o aluno pode ir para qualquer objeto de aprendizagem a qualquer momento. Por sua vez, o autor de um curso pode definir que os exercícios só possam ser feitos depois de se estudar a matéria; ou colocar duas estratégias no curso, uma para o estudo pessoal, mais livre, e outra para uso em sala de aula, mais restrita sob a ótica da ordem de apresentação do conteúdo.

1.1 Trabalhos Relacionados

Não existe algo que vise exatamente a mesma proposta da linguagem Learn, contudo há trabalhos que compartilham algumas das mesmas ideias. Ao que conhecemos, não há muitas linguagens com esses propósitos. Uma, no entanto, é a eLML (*eLesson Markup Language*, [7]). Ela é uma linguagem de marcação baseada em XML que permite a exportação do curso para PDF, LaTeX, ODF, DocBook e outros formatos numa lista em expansão. Contudo, até o presente momento, não dá suporte a MOOCs e não disponibiliza meios de especificar estratégias de ensino, como discorremos acima. Os conteúdos do curso são visualizados apenas na ordem em que foram descritos. Ainda, ao que nos parece, eLML não possui qualquer semântica formal.

Já quanto a propor uma interface única para o desenvolvimento multiplataforma de cur-

sos, os já mencionados SCORM e IMS Learning Design fazem isso através de formatos padronizados a ser lidos pelas plataformas de ensino e de ferramentas próprias para a composição dos cursos. Deles, o SCORM é mais suportado pelos sistemas mais populares enquanto o Learning Design é mais amplo, não apenas apontando os conteúdos, mas modelando todo o processo do aprendizado.

Ainda, quanto ao aprendizado que segue estratégias, ou seja, que tem flexibilidade no aprendizado, mas prevendo restrições um paralelo é o método de ensino usado pelo aplicativo Duolingo [8].

A geração livre de caminhos de aprendizado e a semântica de autômato, duas principais características desta abordagem, são inspiradas por máquinas de Mealy como a semântica de hiperdocumentos [9], hiperautômatos [10] e autômatos não-sequenciais [11]. Autômatos não-sequenciais tem uma estrutura monoidal comutativa que permite a livre geração de ações, ou seja, todos as computações são possíveis por padrão. Foi dos autômatos não-sequenciais de onde veio a ideia de que em Learn "tudo seja possível por padrão".

Por fim, cabe deixar claro que a discussão sobre formas de aprendizado diferentes toca uma área que cabe à psicologia e à pedagogia, algo que nos isentamos de aprofundar neste trabalho.

1.2 Contribuição

A contribuição deste trabalho é a linguagem Learn, dotada de uma semântica formal, e uma série de ferramentas implementadas no sistema de reescrita Maude que permitem a animação e análise dos caminhos de aprendizado do curso, a construção de representações do curso como gramáticas regulares e um transformador para HTML. Assim, no Capítulo 2 são dados os fundamentos necessários ao que será exposto em sequência; o Capítulo 3 apresenta a linguagem e sua semântica; no Capítulo 4 discutimos o conjunto de ferramentas para Learn desenvolvidas no sistema Maude; finalmente, no Capítulo 5 são apresentados exemplos de descrições Maude e da aplicação das ferramentas descritas no Capítulo 4; no Capítulo **??** discutimos os trabalhos relacionados; e no Capítulo 6 damos nossa conclusão e indicamos trabalhos futuros.

Um trabalho de conclusão de curso é uma avaliação total do graduando, complementar às avaliações parciais da disciplina, com o qual se averigua se este alcançou as habilidades necessárias para concluir o curso e receber a titulação própria. Este trabalho corresponde a tal

caráter por tomar as habilidades obtidas na graduação em Ciência da Computação e aplicá-las na pesquisa científica, a fim de solucionar um problema observado. Neste caso, ferramentas computacionais são definidas e desenvolvidas a fim de contribuir na educação.

Capítulo 2

Fundamentação Teórica

Neste capítulo são apresentados os fundamentos teóricos deste trabalho. Na Seção 2.1 são apresentadas algumas definições que serão usadas na formalização da semântica de Learn; na Seção 2.2 são expostas brevemente a Lógica de Reescrita e a linguagem Maude, usadas na formalização de Learn e na sua implementação.

2.1 Gramáticas e Monoides

Nesta seção são dadas algumas definições que serão usadas ao longo do texto.

Devido ao grande uso de gramáticas que será feito, apresentamos primeiro estas e o que a elas se relaciona, a começar pelo conceito fundamental de Palavra.

As definições de gramáticas e linguagens apresentadas a seguir são adaptadas de [9].

Definição 1 (Gramática) *Uma gramática é uma quádrupla ordenada*

$$G = (V, T, P, S)$$

onde

- V *é um conjunto de variáveis (ou não-terminais);*
- T *é um conjunto de terminais;*
- $P : (V \cup T)^+ \to (V \cup T)^*$ *é um conjunto de quádruplas chamadas regras de produção.*
- $S \in V$ *é o símbolo inicial;*

Uma gramática é uma estrutura matemática que gera palavras, as quais são reconhecidas por um autômato. Toda gramática está relacionada a um autômato. Em decorrência disso diz-se que a gramática é um formalismo gerativo e o autômato um formalismo reconhecedor.

A aplicação das regras de produção de uma gramática é chamada derivação, conforme formalmente definida a seguir.

Definição 2 (Relação de Derivação) *Seja G = (V, T, P, S) uma gramática, uma derivação é um par da relação de derivação denotada por $\Rightarrow$, com domínio em $(V \cup T)^+$ e codomínio em $(V \cup T)^*$. Um par $\langle a, b \rangle$ da relação de derivação é representado como*

$$a \Rightarrow b.$$

A relação $\Rightarrow$ é indutivamente definida a seguir:

1. *Para toda produção da forma $S \rightarrow \beta$, pertence à relação de derivação o par*

$$S \Rightarrow \beta$$

2. *Para todo par $\eta \Rightarrow \rho\alpha\sigma$ da relação de derivação, se $\alpha \rightarrow \beta$ é regra de P, então também pertence à relação o par*

$$\eta \Rightarrow \rho\beta\sigma$$

Para a relação de derivação, $\Rightarrow^+$ é o menor fecho transitivo da relação, ou seja, representa uma ou mais aplicações sucessivas da derivação.

A um conjunto de palavras chama-se linguagem. À linguagem relacionada a uma determinada gramática chama-se Linguagem Gerada, que é dada pelo fecho transitivo da relação de derivação. Segue sua definição.

Definição 3 (Linguagem Gerada) *Seja uma linguagem um conjunto de palavras, a linguagem gerada por uma gramática $G = (V, T, P, S)$, $L(G)$, é composta por todas as palavras pertencentes a T^* que podem ser produzidas a partir de S usando as regras de produção, ou seja,*

$$L(G) = \{w \in T^* | S \Rightarrow^+ w\}.$$

Na hierarquia de Chomsky existem quatro tipos de linguagens, relacionadas a quatro tipos de gramáticas. São eles o tipo 0, das linguagens recursivamente enumeráveis, o tipo 1, das linguagens sensíveis ao contexto, o tipo 2, das linguagens livres de contexto, e o tipo 3, das linguagens regulares. A gramática do último tipo, a menos expressiva, é a gramática regular, também chamada de gramática linear.

Definição 4 (Gramática Regular) *Sejam $G = (V, T, P, S)$ uma gramática, $A, B \in V$ variáveis e $w \in T^*$ uma palavra formada de terminais; uma gramática regular (ou linear) é uma G tal que todas as regras em P seguem a forma*

$$A \to w$$

ou uma mesma forma dentre as listadas a seguir:

1. $A \to wB$,
2. $A \to Bw$.

Destaque-se que se $|w| \leq 1$, então a gramática é dita linear unitária. Caso suas produções sejam como na primeira forma, é uma gramática linear unitária à direita, caso contrário é uma gramática linear unitária à esquerda.

A classe de autômatos relacionados às gramáticas regulares é a dos autômatos finitos determinísticos, para os quais há uma conversão direta a partir das gramáticas lineares unitárias à direita e vice-versa.

Três estruturas algébricas importantes para este trabalho são os monóides, os monóides livres e as ordens parciais. As definições de monoide e monoide livre, tal como enunciadas aqui, são adaptadas do terceiro capítulo de [12].

Definição 5 (Monóide) *Um monóide é uma tripla*

$$(A, \oplus, e)$$

onde

- *A é um conjunto, chamado Conjunto Suporte;*
- *$\oplus : A \times A \to A$ é uma operação total e associativa;*
- *e $e \in A$ é um elemento neutro.*

Para nossos fins importa um tipo especial de monoide, chamado Monoide Livremente Gerado ou apenas Monoide Livre.

Definição 6 (Monóide Livre) *Monóide livre é aquele definido como*

$$(A^*, \bullet, \epsilon)$$

e o conjunto suporte agora é A^*, o Fecho de Kleene sobre o conjunto A, agora chamado Conjunto Gerador, $\bullet$ é a operação de concatenação de palavras e ϵ é a palavra vazia.

E, por fim, definimos um tipo especial de relação, a ordem parcial, que é a atribuição de uma ordem sobre os elementos de um conjunto, não necessariamente todos.

Definição 7 (Ordem Parcial) *Uma Relação de Ordem Parcial, denotada por* $(S, \leq)$*, onde S é um conjunto e* $\leq \subseteq S \times S$*, é uma relação reflexiva, antissimétrica e transitiva.*

2.2 Lógica de Reescrita e Maude

Um fundamento importante para embasar este trabalho é a Lógica de Reescrita, a qual explicamos nesta Seção, precedida pela Lógica Equacional de Pertinência e concluindo com sua implementação na linguagem Maude. Durante a Seção também damos alguns conceitos necessários de álgebras. Grande parte desta descrição segue [13].

2.2.1 Lógica Equacional de Pertinência

A Lógica Equacional de Pertinência (do inglês, *Membership Equational Logic* - MEL) [14] é uma generalização da lógica equacional. Ela suporta sortes, subsortes e sobrecarga de operadores. Aqui podemos interpretar sortes como os menores conjuntos que satisfazem os axiomas na teoria equacional. Erros e parcialidades são especificados através de tipos e axiomas de associação condicional.

Uma assinatura na lógica equacional de pertinência é uma tripla (K, Σ, S) (daqui em diante apenas Σ), onde K é um conjunto de tipos, $\Sigma = \{\Sigma_{k_1 \ldots k_n, k}\}_{(k_1 \ldots k_n, k) \in K^* \times K}$ é uma assinatura multi-tipada e $S = \{S_k\} k \in K$ uma família de conjuntos de sortes K-tipados disjuntos dois a dois. O tipo de uma sorte s é denotado por $[s]$. A notação $T_{\Sigma,k}$ e $T_{\Sigma,k}(\vec{x})$ denotam respectivamente o conjunto de Σ-termos ligados com tipo k e de Σ-termos com tipo k sobre variáveis em $\vec{x}$, onde $\vec{x} = \{x_1 : k_1, \ldots, x_n : k_n\}$ é um conjunto de variáveis K-tipadas. A notação $t(\vec{x})$ torna explícito o conjunto de variáveis que aparecem no termo t. Fórmulas atômicas na MEL ou são $t = t'$, onde t e t' são termos do mesmo tipo, ou são declarações associativas da forma $t : s$, onde o termo t é do tipo k e $s \in S_k$.

Sentenças são cláusulas de Horn (conjunção de literais onde no máximo um é positivo) nestas fórmulas atômicas, i.e., sentenças da forma $(\forall \vec{x}) A_0 \ if \ A_1 \wedge \cdots \wedge A_n$, onde cada A_i é ou uma equação ou uma declaração associativa, e $\vec{x}$ é um conjunto de variáveis K-tipadas que contém todas as variáveis que ocorrem em $A_0, A_1, \ldots, A_n$.

$$\frac{t \in T_\Sigma(\vec{x})}{(\forall \vec{x}) t = t}\ \textit{Reflexividade} \qquad \frac{(\forall \vec{x}) t' : s \quad (\forall \vec{x}) t = t'}{(\forall \vec{x}) t : s}\ \textit{Pertinência}$$

$$\frac{(\forall \vec{x}) t' = t}{(\forall \vec{x}) t = t'}\ \textit{Simetria} \qquad \frac{(\forall \vec{x}) t_1 = t_2 \quad (\forall \vec{x}) t_2 = t_3}{(\forall \vec{x}) t_1 = t_3}\ \textit{Transitividade}$$

$$\frac{f \in \Sigma_{k_1 \ldots k_n, k} (\forall \vec{x}) t_i = t_i' \quad t_i, t_i' \in T_{\Sigma, k_i}(\vec{x}) \quad 1 \leq i \leq n}{(\forall \vec{x}) f(t_1, \ldots, t_n) = f(t_1', \ldots, t_n')}\ \textit{Congruência}$$

$$\frac{\begin{array}{c}(\forall \vec{x}) A_0 \ \textit{if}\ A_1 \wedge \ldots \wedge A_n \in E \\ \theta : \vec{x} \to T_\Sigma(\vec{y}) \quad (\forall \vec{y}) \theta(A_i)\ 1 \leq i \leq n\end{array}}{(\forall \vec{y}) \theta(A_0)}\ \textit{Substituição}$$

Figura 2.1: Regras de dedução da Lógica Equacional de Pertinência

Uma teoria na lógica equacional de pertinência é um par (Σ, E), onde E é um conjunto finito de sentenças sobre a assinatura Σ. Termos bem formados são aqueles que provadamente possuem uma sorte. Se não há uma prova, então o termo é mal formado: ele tem um tipo, mas não uma sorte. Por exemplo, seja a soma $+$, a subtração $-$ e a divisão de inteiros $/$ operadores apropriadamente declarados, $3 + 2 : Nat$ e $3 - 4 : Int$, mas $7/0$ é um termo do tipo $[Int]$ sem sorte. A Figura 2.1 mostra o cálculo para a MEL, onde θ é a função de substituição.

Uma teoria (Σ, E) na MEL tem um modelo inicial, denotado $T_{\Sigma/E}$, cujos elementos são classes de equivalência $[t]_E$ de termos sem variáveis. No modelo inicial, igualdade é interpretada como sendo a menor congruência que satisfaz esses axiomas.

2.2.2 Lógica de Reescrita

A Lógica de Reescrita [15] é um modelo computacional e lógico com aplicações em ambos os campos; as teorias de reescrita mantém os dois significados, computacional e lógico. Computacionalmente é um *framework* semântico no qual diferentes modelos computacionais podem ser representados, executados e analisados como teorias de reescrita. Logicamente é um *framework* lógico sobre o qual muitas lógicas diferentes podem ser também representadas, automatizadas e fundamentadas.

Uma teoria de reescrita é uma tripla $\mathcal{R} = (\Sigma, E, R)$, onde Σ é uma coleção de operadores tipados, E é uma coleção de equações e R é uma coleção de regras de reescrita. Uma teoria de

$$\frac{t \in T_\Sigma(\vec{x})}{(\forall \vec{x})t \longrightarrow t}\ \textit{Reflexividade} \qquad \frac{(\forall \vec{x})t_1 \longrightarrow t_2 \quad (\forall \vec{x})t_2 \longrightarrow t_3}{(\forall \vec{x})t_1 \longrightarrow t_3}\ \textit{Transitividade}$$

$$\frac{E \vdash (\forall \vec{x})t = u \quad (\forall \vec{x})u \longrightarrow u' \quad E \vdash (\forall \vec{x})u' = t'}{(\forall \vec{x})t \longrightarrow t'}\ \textit{Igualdade}$$

$$\frac{f \in \Sigma_{k_1 \ldots k_n,k}(\forall \vec{x})t_i \longrightarrow t_i' \quad t_i, t_i' \in T_{\Sigma,k_i}(\vec{x}) \quad 1 \leq i \leq n}{(\forall \vec{x})f(t_1, \ldots, t_n) \longrightarrow f(t_1', \ldots, t_n')}\ \textit{Congruência}$$

$$\frac{\begin{array}{c}(\forall \vec{x})\lambda : t \to t' \textit{ if } \bigwedge_{i \in I} p_i = q_i \wedge \bigwedge_{j \in J} w_j : s_j \wedge \bigwedge_{l \in L} t_l \to t_l' \in R \\ \theta : \vec{x} \to T_\Sigma(\vec{y}) \quad (\forall \vec{y})\theta(t_l) \longrightarrow \theta(t_l')\ l \in L \\ E \vdash (\forall \vec{y})\theta(p_i) = \theta(q_i)\ i \in I \quad E \vdash (\forall \vec{y})\theta(w_j) : s_j\ j \in J\end{array}}{(\forall \vec{y})\theta(t) \longrightarrow \theta(t')}\ \textit{Substituição}$$

Figura 2.2: Regras de dedução da Lógica de Reescrita

reescrita tem duas partes: (Σ, E), a parte estática, que é uma teoria equacional de pertinência e descreve a estrutura de um estado no sistema especificado; e R, a parte dinâmica, que descreve as transições entre estados do sistema, às quais seguem a forma geral

$$\lambda : (\forall \vec{x})t \to t' \textit{ if } \bigwedge_{i \in I} p_i = q_i \wedge \bigwedge_{j \in J} w_j : s_j \wedge \bigwedge_{l \in L} t_l \to t_l',$$

onde λ é um rótulo, $p_i = q_i$ e $w_j : s_j$ são fórmulas atômicas na lógica equacional de pertinência para $i \in I$ e $j \in J$, e para os tipos apropriados k e $k_l, t, t' \in T_{\Sigma,k}(\vec{x})$, e $t_l, t_l' \in T_{\Sigma,k_l}(\vec{x})$ para $l \in L$. Variáveis livres, ou seja, aquelas não quantificadas numa expressão, são permitidas nas condições desde que adicionadas incrementalmente [16] em termos de variáveis no lado esquerdo da regra e também nas condições.

As regras de dedução da Lógica de Reescrita são dadas na Figura 2.2. Dados dois estados $[u], [v] \in T_{\Sigma/E,k}$, $[v]$ pode ser alcançado a partir de $[u]$ por algumas computações concorrentes possivelmente complexas se, e somente se puder ser provado que $u \longrightarrow v$ na lógica.

Aspectos meta-lógicos. Conforme no Capítulo 11 de [17] existe uma teoria de reescrita universal $\mathcal{U}$ na qual é possível representar qualquer teoria de reescrita $\mathcal{R}$, incluindo a própria teoria $\mathcal{U}$, como um termo $\overline{\mathcal{R}}$, quaisquer termos t, t' em $\mathcal{R}$ como termos $\overline{t}$, $\overline{t'}$, e qualquer par ($\mathcal{R}$, t) como um termo $\langle \overline{\mathcal{R}}, \overline{t} \rangle$, de modo que tenha-se a seguinte equivalência

$$\mathcal{R} \vdash t \longrightarrow t' \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \to \langle \overline{\mathcal{R}}, \overline{t'} \rangle.$$

Sendo $\mathcal{U}$ representável em si mesma é possível aplicar essa equivalência quantas vezes forem desejadas, aplicando o processo acima repetidas vezes, como a seguir, subindo a cada passo o nível da dedução:

$$\mathcal{R} \vdash t \to t' \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \to \langle \overline{\mathcal{R}}, \overline{t'} \rangle \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t} \rangle} \rangle \to \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t'} \rangle} \rangle \ldots$$

2.2.3 Maude

Maude é uma linguagem de programação de alto nível que implementa a Lógica de Reescrita e, por isso, suporta computações equacionais e de reescrita. Na especificação de um sistema em Maude há a separação entre as partes estática e dinâmica. A parte estática é representada através de módulos funcionais, enquanto a parte dinâmica o é através de módulos de sistema. Uma computação num sistema de transição é capturada pela reescrita de termos, enquanto estados do sistema, segundo as equações.

Sua semântica operacional é a simplificação equacional. A reescrita é aplicada até que a forma canônica seja obtida e um dado termo não possa mais ser reescrito. Equações são usadas para definir funções sobre dados estáticos assim como propriedades de estados.

Módulos de sistema em Maude correspondem a teorias de reescrita. As regras de reescrita devem ser coerentes com as equações, o que significa que a intercalação da reescrita com regras e da reescrita com equações não perderá computações de reescrita, ou seja, não deixará de cumprir uma reescrita que de outro modo teria sido possível antes de dar um passo de dedução equacional. Assumindo a coerência, Maude sempre reduz para a forma canônica usando E antes de aplicar qualquer regra em R.

Exemplos de programação em Maude. No exemplo a seguir ilustramos a sintaxe dos módulos de Maude representando o processo de geração de palavras através de uma gramática, usando a reescrita. Para isto tomamos como exemplo uma gramática onde A, B e C são os elementos do alfabeto da linguagem, que é composta de quaisquer palavras geradas a partir desse alfabeto, exceto aquelas que possuam símbolos iguais contíguos.

O módulo funcional GRAMMAR declara o sorte *Word* que representa as palavras geradas pela gramática, sendo epsilon a palavra vazia. V e T são sortes que designam as variáveis e os terminais da gramática; enquanto *Initial* é o sorte do estado inicial, declarado como o operador S.

```
1 fmod GRAMMAR is
```

```
  sorts V Word Initial T .
  subsorts V Initial T < Word .
  op epsilon : → Word .
  op S : → Initial .
  op __ : Word Word → Word [assoc id: epsilon] .
endfm
```

Listing 2.1: Módulo funcional GRAMMAR

Já o módulo de sistema ABC-GRAMMAR descreve a gramática descrita acima. As regras de reescrita implementam exatamente as regras de produção da gramática.

```
mod ABC−GRAMMAR is
  inc GRAMMAR .
  ops A B C : → T .
  ops v0 v1 v2 : → V .
  rl S ⇒ epsilon . rl S ⇒ A . rl S ⇒ A v0 . rl S ⇒ B . rl S ⇒ B v1 . rl S ⇒ C . rl S ⇒ C v2 .
  rl v0 ⇒ B . rl v0 ⇒ B v1 . rl v0 ⇒ C . rl v0 ⇒ C v2 .
  rl v1 ⇒ A . rl v1 ⇒ A v0 . rl v1 ⇒ C . rl v1 ⇒ C v2 .
  rl v2 ⇒ A . rl v2 ⇒ A v0 . rl v2 ⇒ B . rl v2 ⇒ B v1 .
endm
```

Listing 2.2: Módulo de sistema ABC-GRAMMAR

Executando o interpretador de Maude podemos usar o comando *search* para fazer a geração de palavras. Apresentamos na Listing 2.3 apenas as primeiras 5 palavras, na ordem em que são geradas, mas o sistema é capaz de gerar qualquer palavra da linguagem gerada pela gramática.

```
search in ABC−GRAMMAR : S ⇒! W:Word .

Solution 1 (state 1)
states: 8 rewrites: 7 in 0ms cpu (0ms real) (~ rewrites/second)
W:Word ⟶ epsilon

Solution 2 (state 2)
states: 8 rewrites: 7 in 0ms cpu (0ms real) (~ rewrites/second)
W:Word ⟶ A

Solution 3 (state 4)
states: 12 rewrites: 11 in 0ms cpu (0ms real) (~ rewrites/second)
W:Word ⟶ B

Solution 4 (state 6)
states: 16 rewrites: 15 in 0ms cpu (0ms real) (~ rewrites/second)
W:Word ⟶ C

Solution 5 (state 8)
```

```
20 states: 20 rewrites: 19 in 0ms cpu (0ms real) (~ rewrites/second)
21 W:Word ⟶ A B
```

Listing 2.3: Geração de palavras

Para verificar se uma palavra pertence à linguagem gerada basta usar novamente o comando *search* e verificar se é possível produzi-la a partir do símbolo inicial. Na primeira execução mostrada na Listing 2.4 a palavra ABAC é reconhecida, já que é encontrada uma solução, enquanto na segunda ABAA não é, pois não é encontrada nenhuma. Note que o processo de geração é infinito, portanto a busca de aceitação foi limitada pela profundidade, aqui equivalente ao tamanho máximo das palavras. Atente-se também a que a busca de aceitação foi limitada pela profundidade, aqui equivalente ao tamanho máximo das palavras, ou seja, apenas buscar nas palavras de tamanho menor ou igual ao dado, porque, se a linguagem gerada for infinita, a geração é infinita, uma vez que todas as possíveis palavras são geradas. Não se deve confundir a geração das infinitas palavras com o complemento fechado das linguagens regulares.

```
1  search [, 4] in ABC−GRAMMAR : S ⇒∗ A B A C .
2
3  Solution 1 (state 46)
4  states: 47 rewrites: 46 in 0ms cpu (0ms real) (~ rewrites/second)
5  empty substitution
6
7  No more solutions.
8  states: 92 rewrites: 91 in 0ms cpu (0ms real) (~ rewrites/second)
9  ==========================================
10 search [, 4] in ABC−GRAMMAR : S ⇒∗ A B A A .
11
12 No solution.
13 states: 92 rewrites: 91 in 0ms cpu (0ms real) (~ rewrites/second)
```

Listing 2.4: Aceitação de palavras

Exemplos de meta-programação em Maude. Em Maude a teoria universal $\mathcal{U}$ é implementada pelo módulo META-LEVEL, o qual define estruturas de dados para manipular termos e módulos no meta-nível, ou seja, no nível de teorias representadas em $\mathcal{U}$.

Tal como é perceptível pela teoria, para subir um nível no META-LEVEL de Maude é necessário apenas a representação do módulo desejado com a sintaxe apropriada. Nesse sentido Maude traz as chamadas funções de meta-nível (do inglês, *descent functions*), como *upModule* e *upTerm*, que tomam um termo e produzem sua meta-representação. Por outro lado, há funções complementares, como *downTerm*, que representa o termo um nível abaixo. Há ainda funções

que operam em módulos descritos no meta-nível, como *metaReduce* e *metaSearch*, que são operações parciais que aplicam as mesmas estratégias de suas correspondentes em nível de objeto (o nível mais baixo, em oposição ao meta-nível), as quais são comandos de Maude.

Abaixo demonstramos o mesmo comando feito na Listing 2.3 em nível de objeto, agora representado no META-LEVEL. O resultado é a última palavra gerada, dentro da limitação imposta pela função.

```
1 mod META−LEVEL−GRAMMAR is inc META−LEVEL . inc ABC−GRAMMAR . endm
2 ==========================================
3 reduce in META−LEVEL−GRAMMAR : metaSearch(upModule('ABC−GRAMMAR, false), upTerm(S), upTerm(W:Word), nil, '!, 2, 4) .
4 rewrites: 23 in 0ms cpu (0ms real) (~ rewrites/second)
5 result ResultTriple: {'__['A.T,'B.T],'Word, 'W:Word ← '__['A.T,'B.T]}
```

Listing 2.5: Geração de palavras no meta-nível

Analogamente, na Listing 2.6 aplicamos no meta-nível o mesmo comando da Listing 2.4, para o reconhecimento de palavras. Dada uma palavra como argumento, se ela for encontrada a partir da palavra inicial, então ela foi aceita, ou seja, pertence à linguagem gerada pela gramática. Se o retorno for uma falha então a palavra não pertence à linguagem e não foi aceita.

```
1 reduce in META−LEVEL−GRAMMAR : metaSearch(upModule('ABC−GRAMMAR, false), upTerm(S), upTerm(A B A C), nil, '!, 4, 0) .
2 rewrites: 96 in 0ms cpu (0ms real) (~ rewrites/second)
3 result ResultTriple: {'__['A.T,'B.T,'A.T,'C.T],'Word,(none).Substitution}
4 ==========================================
5 reduce in META−LEVEL−GRAMMAR : metaSearch(upModule('ABC−GRAMMAR, false), upTerm(S), upTerm(A B A A), nil, '!, 4, 0) .
6 rewrites: 119 in 0ms cpu (0ms real) (~ rewrites/second)
7 result ResultTriple?: (failure).ResultTriple?
```

Listing 2.6: Aceitação de palavras no meta-nível

A saída do *metaSearch* pode ser levada para o nível de objeto usando a composição das funções *getTerm* e *downTerm*. A aplicação da função *downTerm* sobre o primeiro componente de um *ResultTriple* (a saída do *metaSearch*) produz a representação em nível objeto de um meta-termo ou o segundo argumento, quando ele falha em produzir a representação em nível objeto do primeiro argumento.

```
1 reduce in META−LEVEL−GRAMMAR : downTerm(getTerm(metaSearch(upModule('ABC−GRAMMAR, false), upTerm(S), upTerm(W:
      Word), nil, '!, 2, 4)), error:Word) .
2 rewrites: 25 in 4ms cpu (0ms real) (6250 rewrites/second)
3 result Word: A B
```

Listing 2.7: Geração de palavras no meta-nível levada ao nível de objeto

Capítulo 3

Learn

Neste capítulo apresentamos a linguagem Learn, primeiramente através de alguns exemplos, depois expondo sua formalização. Inicialmente, para ilustrar a intuição, tomemos a Figura 3.1 como um exemplo do comportamento de um curso em Learn. Note que o autômato associado reconhece a linguagem gerada pela gramática descrita na Listing 2.2.

Supomos no exemplo o caso de um curso sem estratégia, ou seja, onde qualquer ordem de estudo é aceitável, e os objetos de aprendizagem (i.e. os conteúdos para estudo) são A, B e C quaisquer. No autômato da Figura 3.1 os estados representam o conjunto de objetos que podem ser escolhidos para a próxima computação, que são todos no estado inicial e qualquer conteúdo a menos do último escolhido nos outros casos. As transições denotam uma escolha que foi feita. Assim, como esperado, escolhendo-se A a partir do estado inicial, a transição é feita para o estado $\{B, C\}$, onde não se pode escolher A novamente. Neste contexto, pode-se entender que a adição de uma estratégia imporia restrições sobre as transições do autômato, afetando seu comportamento.

Na Seção 3.1 são mostrados (i) o exemplo mais simples e mais geral onde é declarado um conjunto de objetos de aprendizagem e um curso que ensina um subconjunto deles; (ii) o segundo exemplo declara ordens sequenciais entre os objetos de aprendizagem ensinados no curso por meio de duas estratégias de ensino; e (iii) um curso com uma estratégia de ensino que faz uso de conjuntos genéricos de objetos de aprendizagem. Já na Seção 3.2 expomos uma semântica formal dada através de gramáticas para os cursos em Learn e suas estratégias.

Os exemplos usados na Seção 3.1 são retirados dos Capítulos 2 e 3 de [9].

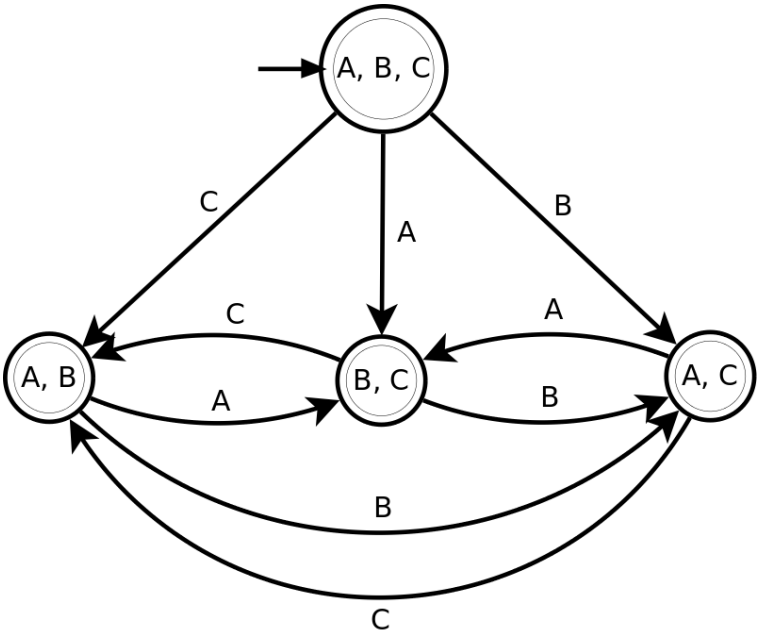


Figura 3.1: Intuição para um curso em Learn

3.1 Sintaxe e Semântica informal

Nesta seção apresentamos Learn através de exemplos, demonstrando a sua sintaxe e por meio dela, informalmente, a sua semântica.

Curso sem estratégia. Uma descrição em Learn é composta da declaração do curso seguida da declaração dos objetos de aprendizagem, que são os menores elementos de Learn. A princípio pode-se entender um curso como um simples conjunto de objetos de aprendizagem a ser apresentado.

A declaração de um objeto é dada conforme a seguir, contendo um identificador, um título e, facultativamente, atributos como texto ou uma imagem, que constroem o material de estudo propriamente.

```
1 learning object < 'int3 > has
2     title ``Introducao ao Capitulo 3''
3     image ``auto.png''
4     text ``O estudo das linguagens regulares . . .''
```

Segue então a declaração do curso. Ele possui um nome e, necessariamente, são listados os objetos de aprendizagem que o compõe, após a palavra chave *teaches*. Conforme dito anteriormente, na ausência de uma estratégia o curso é apenas um conjunto de objetos para serem estudados e, sendo assim, não existindo uma ordem de ensino pré-estabelecida, todos os possí-

veis caminhos de aprendizagem são permitidos. A título de ilustração, é possível começar pela introdução do capítulo, $int3$, seguir para uma das seções com a explicação do conteúdo, $sec3\_1$ ou $sec3\_2$, consultar os exemplos, $fig3\_2$, e depois fazer os exercícios, $ex3\_1$ e $ex3\_2$, como também começar pelos exemplos, tentar os exercícios para só então olhar a explicação e depois voltar aos mesmos exercícios.

```
course on ''Linguagens Formais''
    teaches < 'int3 >, < 'sec3_2 >, < 'sec3_1 >, < 'fig3_2 >, < 'ex3_1 > and < 'ex3_2 >
```

Listing 3.1: Exemplo sem estratégia

Curso com duas estratégias. Um curso com uma estratégia é como o apresentado previamente munido de uma ordem parcial sobre os objetos de aprendizagem. É possível definir mais de uma estratégia de ensino. Neste caso, considera-se a união disjunta das estratégias. Esta ordem restringe o acesso a um determinado objeto criando para ele um conjunto não vazio de pré-requisitos que devem ser estudados antes que se possa alcançar o referido objeto. (Note que o caso sem estratégias é exatamente aquele em que o conjunto de pré-requisitos é vazio.) Tomando uma regra do caso abaixo, na estratégia *class* o objeto *sec3_1* deve ser estudado antes do *sec3_2*, sendo assim, caso ainda não se tenha passado por *sec3_1*, *sec3_2* não estará disponível.

Note que no caso de haver mais de uma estratégia elas não se confundem. Entenda que o aluno opta por seguir o curso segundo uma determinada estratégia e enquanto está em uma ignora as demais.

```
course on ''Linguagens Formais''
    teaches < 'sec2_1 >, < 'sec2_2 >, < 'sec2_4 >, < 'int3 >, < 'sec3_1 >, < 'sec3_2 >, < 'sec3_3 >, < 'fig3_2 >, < 'sec3_7 >, < 'ex3_1 >
        and < 'ex3_2 >
with
    teaching strategy < 'class >
        < 'int3 > before < 'sec3_1 >,
        < 'sec3_1 > before < 'sec3_2 >,
        < 'sec3_2 > before < 'fig3_2 >,
        < 'fig3_2 > before < 'sec2_1 >,
        < 'sec2_1 > before < 'sec2_2 >,
        < 'sec2_2 > before < 'sec3_3 >,
        < 'sec3_3 > before < 'sec2_4 >,
        < 'sec2_4 > before < 'sec3_7 >,
        < 'sec3_7 > before < 'ex3_1 >,
        < 'ex3_1 > before < 'ex3_2 >
    teaching strategy < 'book >
        < 'sec2_1 > before < 'sec2_2 >,
        < 'sec2_2 > before < 'sec2_4 >,
        < 'sec2_4 > before < 'int3 >,
```

```
        < 'int3 > before < 'sec3_1 >,
        < 'sec3_1 > before < 'sec3_2 >,
        < 'sec3_2 > before < 'sec3_3 >,
        < 'sec3_3 > before < 'fig3_2 >,
        < 'fig3_2 > before < 'sec3_7 >,
        < 'sec3_7 > before < 'ex3_1 >,
        < 'ex3_1 > before < 'ex3_2 >
```

Listing 3.2: Exemplo com duas estratégias

Curso que define um subconjunto de objetos de aprendizagem. Tendo Learn um propósito específico entendemos que podem haver alguns subconjuntos de objetos de aprendizagem que sejam discrimináveis, dos quais visualizamos os exercícios. No contexto de um curso os exercícios são a aplicação do conteúdo e sua revisão, tem um caráter especial diante dos outros objetos e podem requerer estratégias sobre eles como um todo. Neste exemplo esse subconjunto é explicitado e é construída uma estratégia sobre ele, impondo que os exercícios só poderão ser feitos após todos os outros objetos terem sido estudados..

```
course on ''Linguagens Formais''
    teaches < 'int3 >, < 'sec3_2 >, < 'sec3_1 >, < 'fig3_2 >, < 'ex3_1 > and < 'ex3_2 >
with
    exercises < 'ex3_1 > and < 'ex3_2 >
    teaching strategy < 'ex >
        exercises after all but the exercises
```

Listing 3.3: Exemplo com definição de subconjunto

Vale ressaltar a diferença para o exemplo anterior, onde as construções são explícitas sobre os objetos de aprendizagem, apontando para cada determinado objeto seus predecessores específicos. Neste exemplo, a estratégia é descrita através de operações sobre conjuntos.

3.2 Semântica Formal

Learn é projetada para desde o princípio apresentar um comportamento matematicamente preciso. Para tal, sua semântica formal é apresentada nesta Seção e sobre ela demonstra-se a expressividade de um curso Learn. Ainda, Learn enxerga um curso como uma estrutura formal, ou seja, um curso descrito em Learn é também a formalização da noção de curso e do processo de estudo.

Uma descrição em Learn é uma tupla

$$L = (C, O, \varsigma)$$

onde C é o identificador do curso, O é um conjunto de identificadores de objetos de aprendizagem e ς é a união disjunta de ordens parciais $(D, \leq)$ onde cada uma denota uma estratégia de ensino, com $D \subseteq O$. Uma relação $\preceq \subseteq D^* \times D$ pode ser derivada diretamente de $(D, \leq)$, denotando o conjunto de predecessores de um dado objeto de aprendizagem.

Dada a ordem parcial, tomamos dois subconjuntos de O: O_{pred}, o conjunto dos identificadores de objetos de aprendizagem no qual cada objeto s é predecessor de algum outro objeto o em O, e O_{suc}, o conjunto dos identificadores de objetos de aprendizagem no qual cada objeto s é sucessor de algum outro objeto o em O. Ainda, faremos uso da relação $\preceq$, que indica que um elemento é predecessor de outro (ou ele mesmo).

Definição 8 (Gramática Learn) *Cada estratégia de L é associada a uma gramática regular*

$$G_L = (V, T, P, S)$$

onde $V \subset O \times 2^{O_{pred}}$, $T = O$ e as produções em P são conforme a seguir.

As produções iniciais (a partir da palavra inicial) são $S \to \epsilon$ e

$$\forall o \in O - O_{suc}, S \to o \mid o\ (o, \{o\}), \textit{ se o} \in O_{pred}$$

$$S \to o \mid o\ (o, \emptyset), \textit{ caso contrário}$$

e as demais são

$$\forall o, o' \in O, \forall p \in 2^{O_{pred}}, (o, p) \to o' \mid o'\ (o', p \cup o'), \textit{se o'} \in O_{pred}$$

$$(o, p) \to o' \mid o'\ (o', p), \textit{caso contrário}$$

tal que $\forall e \preceq o, e \in p; \forall \pi \in p, \forall e \preceq \pi, e \in p;\ o \in O_{pred} \leftrightarrow o \in p;\ o \neq o';\ \forall e \preceq o', e \in p.$

Note que ambos os padrões apresentado acima, na definição, indicam a geração de pares de regras com o lado esquerdo idêntico, uma cujo lado direito é formado por um terminal, outra cujo lado direito é formado por um terminal seguido de um não-terminal.

Exemplo. Seja um curso para o qual $O = \{A, B, C\}$ e com a estratégia onde $B \leq C$.

As produções da gramática gerada serão:

$S \to \epsilon \mid A \mid A\ (A, \emptyset) \mid B \mid B\ (B, \{B\})$
$(A, \emptyset) \to B \mid B\ (B, \{B\})$
$(A, \{B\}) \to B \mid B\ (B, \{B\}) \mid C \mid C\ (C, \{B\})$
$(B, \{B\}) \to A \mid A\ (A, \{B\}) \mid C \mid C\ (C, \{B\})$
$(C, \{B\}) \to A \mid A\ (A, \{B\}) \mid B \mid B\ (B, \{B\})$

Os pares de V, como por exemplo (B, {B}), significam o último objeto escolhido e o conjunto de todos os predecessores já produzidos. A semântica é que só se pode gerar um objeto distinto do último gerado e cujos predecessores já foram produzidos. Feito isso, nova informação é acrescentada no estado alcançado.

Consequentemente, não podem haver produções que gerem C a partir de S porque C só pode ser gerado uma vez que um B também tenha sido. Pela mesma razão, não é possível produzir um não-terminal cujo segundo elemento não seja superconjunto daquele presente no não-terminal à esquerda da regra, pois significaria a perda de informação.

Finalmente, as gramáticas geradas serão sempre regulares porque seus autômatos correspondentes são restrições sobre monoides livres, que aumentam o número de estados e transições. Mais precisamente, são gramáticas lineares unitárias à direita.

A partir dessa formalização, definimos a noção de caminho de aprendizagem.

Definição 9 (Caminho de Aprendizagem) *Um caminho de aprendizagem de L é uma palavra da Linguagem Gerada por G_L construída de acordo com a Definição 8.*

E definimos, por fim:

Definição 10 (Semântica de um Curso) *A gramática associada a um curso é a união disjunta das gramáticas de suas estratégias.*

Donde se destaca o seguinte Fato:

Fato 1 *A gramática associada a um curso também é regular uma vez que a as gramáticas regulares são fechadas para a união.*

Capítulo 4

Maude Learn Toolkit

Apresentamos na Seção 2.2 a Lógica de Reescrita e a linguagem Maude. Como dito anteriormente, Learn foi pensada para ter uma semântica formal atrelada à sua sintaxe a fim de termos um significado bem definido para suas especificações. Lógica de reescrita e Maude são utilizados na implementação de diferentes ferramentas para a animação, análise e implantação de cursos Learn. A razão maior da escolha é dada pela pelas características meta-lógicas implementadas pelas funções de meta-nível em Maude. Seria possível utilizar outra linguagem, por exemplo Prolog. Todavia, Prolog é apenas uma linguagem de programação e seria necessário um ambiente externo. Já Maude é uma implementação da Lógica de Reescrita e devido a isso computações em Maude, isto é, reescritas, denotam deduções na lógica; ainda, a meta-programação dispensa a necessidade de ambiente externo, mantendo ainda a simplicidade e a concisão. Isto faz com que descrições em Learn sejam denotadas por teorias de reescrita implementadas em módulos de sistema, tornando-as executáveis.

Neste capítulo mostraremos o Maude Learn Toolkit, uma série de ferramentas implementadas no sistema de reescrita Maude que permitem a animação e análise dos caminhos de aprendizado do curso, a construção de representações do curso como gramáticas regulares e um transformador para HTML, cuja principal peça é o transformar de descrições Learn para gramáticas. Primeiramente, a Seção 4.1 detalha o analisador sintático, capaz de reconhecer uma descrição Learn; a seguir, sobre ele, é apresentado na Seção 4.2 um transformador que converte descrições Learn em gramáticas regulares implementadas como módulos em Maude. Em seguida, descrevemos uma implementação alternativa, criada ao longo do desenvolvimento do projeto a fim de entender com clareza o comportamento intuído para a linguagem. Na Seção 4.3 discutimos um transformador, LearnMaude, que converte descrições Learn em teorias de re-

escrita implementadas em Maude e na Seção 4.4 uma extensão de LearnMaude, LearnHTML, que a partir destas teorias gera páginas web. Estas são usadas como prova de conceito para a abordagem formal para construção de cursos online apresentada neste trabalho. Vale ressaltar que os documentos HTML produzidos pelo transformador LearnHTML podem ser inseridas na plataforma Moodle e executadas dentro deste ambiente.

Esta seção visa discutir a estrutura e os principais aspectos do protótipo do transformador de Learn, disponível em `https://github.com/HugoFarias/learn/` e no Apêndice B.

4.1 Analisador Sintático

Nesta seção descrevemos o módulo funcional Maude LEARN-SIG, que implementa um analisador sintático para a linguagem Learn. A descrição da sintaxe de Learn como assinatura ordenada-sortida $\Sigma_{LEARN-SIG}$ permite a descrição da sintaxe de uma linguagem livre de contexto.

Primeiramente, segundo a descrição dada por LEARN-SIG, entendemos que uma descrição Learn, representado pelo sorte LearnCourse, é a concatenação da declaração do curso, com sua estratégia, com as declarações dos objetos de aprendizagem. A declaração de um curso, como mostrada abaixo no seu caso geral, é composta de um identificador para o curso, um conjunto de identificadores de seus objetos de aprendizagem e uma ou mais estratégias, contidas em CourseSpecs. A estratégia é construída contendo um identificador e um conjunto de relações de ordem, SetStr, que indica as dependências entre os objetos de aprendizagem do curso. A sorte de um conjunto de declarações de estratégias de ensino, TeachStrDecl, é um subsorte de CourseSpecs, entendendo que é um entre outros tipos de especificações de um curso. Já a declaração de um objeto de aprendizagem traz, além de seu identificador, um título e um conjunto de atributos.

```
1 op _ _ : CourseDecl LearnObjDecls → LearnCourse .
2 op course on_teaches_and_with_ : CourseID SetObj LearnID CourseSpecs → CourseDecl .
3 op teaching strategy_ _ : LearnID SetStr −> TeachStrDecl [prec 20] .
4 op learning object_has title_ _ : LearnID String SetLOAttrib −> LearnObjDecl [prec 50] .
```

A partir destas construções fundamentais da sintaxe de Learn decorre que a descrição de um curso seja conforme a forma geral a seguir, que resume a estrutura usada nos exemplos da Seção 3.1.

```
1 course on ‘‘Linguagens Formais’’ teaches < ’sec2_1 >, . . . and < ’ex3_2 > with . . .
2     teaching strategy < ’ex > . . .
3 learning object < ’int3 > has title . . .
```

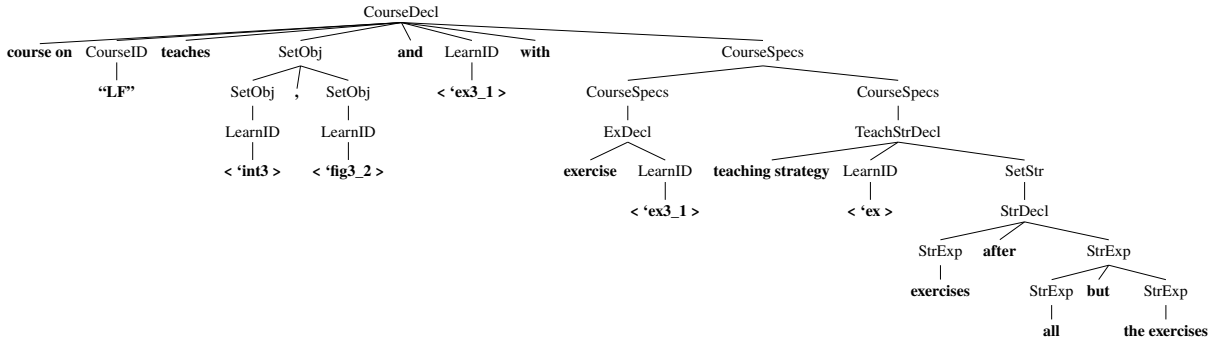


Figura 4.1: Árvore sintática de uma simplificação da descrição da Listing 3.3

A fim de demonstrar o funcionamento do analisador sintático de Learn mostramos a seguir uma árvore sintática para uma simplificação da descrição dada na Listing 3.3, com menos objetos de aprendizagem.

Cabe aqui sublinhar que a concatenação de um objeto da sorte CourseDecl com um da sorte LearnObjDecl representa uma descrição completa do curso. Ou seja, ao exposto na Listing e na árvore acrescentam-se as definições do conteúdo dos objetos de aprendizagem.

4.2 Transformador de Learn para Gramáticas Regulares

Este transformador é uma implementação em Maude da função que converte um curso descrito em Learn em gramáticas regulares conforme o modelo apresentado na Seção 3.2.

O módulo principal é o GRAMMAR-TRANSFORM, que implementa uma meta-função, transform-in-grammar, que tem como argumento uma especificação em Learn, um termo em $\mathcal{T}_{LEARN-SIG}$, e gera o conjunto de gramáticas associado ao curso descrito em um meta-módulo. A implementação das gramáticas é uma modificação da apresentada na Listing 2.1.

```
1 op transform−in−grammar : LearnCourse → SModule .
```

A fim de prover comandos simples sobre o resultado que encapsulem as chamadas de funções de Maude usamos um módulo de sistema que estende o GRAMMAR-TRANSFORM chamado LEARN-CMD. A seguir apresentamos as funções contidas em LEARN-CMD.

```
1 op getGrammar : LearnCourse Qid → ProductionSet .
2 op getGrammar : Module Qid → ProductionSet .
```

Em primeiro lugar, getGrammar, que dados um curso em Learn (ou o módulo de sistema que descreve sua gramática) e a estratégia desejada, imprime as regras de produção da gramática

associada. Para um curso sem estratégia, o identificador é *default*. Por razões de implementação os pares ordenados estão entre colchetes, ao invés de parênteses.

```
op getLabeledGrammar : LearnCourse Qid → ProductionSet .
op getLabeledGrammar : Module Qid → ProductionSet .
```

Alternativamente, getLabeledGrammar faz o mesmo que getGrammar, mas troca os não-terminais por rótulos formados pelo caractere 'v' concatenado a um número natural.

```
op genWord : LearnCourse Qid Nat → Word .
op genWord : Module Qid Nat → Word .
```

O comando genWord gera palavras de acordo com um índice.

```
op hasWord : LearnCourse Qid Word → Bool .
op hasWord : Module Qid Word → Bool .
```

E o comando hasWord verifica se a palavra dada pertence à linguagem gerada pela gramática passada como parâmetro.

Exemplos. Exemplificamos a seguir os comandos dados pelo módulo CMD do transformador para gramáticas.

Na Listing 4.2 apresentamos um exemplo da geração de uma gramática para uma descrição Learn de um curso que tem como objetos de aprendizagem A, B e C, e a única estratégia de ensino define que B deve ser estudado antes de C, ou seja, é o mesmo exemplo dado na Seção 3.2. Aqui, eg0 é uma constante da sorte LearnCourse que contém a descrição em Learn do curso. Esta e outras constantes para os exemplos usados aqui são descritas em um módulo chamado TEST.

```
course on "TEST"
  teaches < 'A >, < 'B > and < 'C >
with
  teaching strategy < 's >
    < 'B > before < 'C >
```

Listing 4.1: Exemplo simples em Learn

Usamos este exemplo por simplicidade, devido à extensão das gramáticas associadas aos cursos.

```
reduce in TEST : getGrammar(eg0, 's) .
rewrites: 583 in 0ms cpu (0ms real) (~ rewrites/second)
result ProductionSet: (S → 'A)
(S → 'B)
(S → epsilon)
```

```
(S → 'A ['A,{}])
(S → 'B ['B,{'B}])
(['A,{}] → 'B)
(['A,{}] → 'B ['B,{'B}])
(['A,{'B}] → 'B)
(['A,{'B}] → 'C)
(['A,{'B}] → 'B ['B,{'B}])
(['A,{'B}] → 'C ['C,{'B}])
(['B,{'B}] → 'A)
(['B,{'B}] → 'C)
(['B,{'B}] → 'A ['A,{'B}])
(['B,{'B}] → 'C ['C,{'B}])
(['C,{'B}] → 'A)
(['C,{'B}] → 'B)
(['C,{'B}] → 'A ['A,{'B}])
['C,{'B}] → 'B ['B,{'B}]
```

Listing 4.2: Geração da gramática associada ao exemplo 4.1

Alternativamente, ao usar o comando getLabeledGrammar os elementos de V serão renomeados.

```
reduce in TEST : getLabeledGrammar(eg0, 's) .
rewrites: 677 in 0ms cpu (1ms real) (~ rewrites/second)
result ProductionSet: ('v0 → 'B)
('v0 → 'B 'v1)
('v1 → 'A)
('v1 → 'C)
('v1 → 'A 'v2)
('v1 → 'C 'v3)
('v2 → 'B)
('v2 → 'C)
('v2 → 'B 'v1)
('v2 → 'C 'v3)
('v3 → 'A)
('v3 → 'B)
('v3 → 'A 'v2)
('v3 → 'B 'v1)
(S → 'A)
(S → 'B)
(S → epsilon)
(S → 'A 'v0)
S → 'B 'v1
```

Listing 4.3: Geração da gramática associada ao exemplo 4.1 com rótulos

Na Listing 4.4 exemplificamos o comando genWord, para geração de palavras, sobre o exemplo da Listing 3.2. Geramos uma palavra na estratégia *class*, mostrando um caminho de

aprendizagem. Como esperado, esse caminho é linear e começa por *int3*, que é o único objeto de aprendizagem da estratégia sem uma restrição.

```
1 reduce in TEST : genWord(eg2, 'class, 7) .
2 rewrites: 21811094 in 20088ms cpu (20088ms real) (1085777 rewrites/second)
3 result Word: 'int3 'sec3_1 'sec3_2 'fig3_2
```

Listing 4.4: Geração de palavras para o exemplo 3.2

Finalmente, na Listing 4.5 é dado um exemplo da aceitação de palavras sobre o curso descrito na Listing 3.3. No primeiro exemplo a palavra é aceita, já no segundo, que viola a restrição da estratégia de exercícios só serem alcançados após todos os demais objetos, a palavra não é aceita, ou seja, não pertence à linguagem gerada pela gramática e, consequentemente, o caminho de aprendizagem dado por ela não é possível.

```
1 reduce in TEST : hasWord(eg3, 'ex, 'int3 'sec3_1 'sec3_2 'fig3_2) .
2 rewrites: 49646 in 20ms cpu (19ms real) (2482300 rewrites/second)
3 result Bool: true
4 ==========================================
5 reduce in TEST : hasWord(eg3, 'ex, 'int3 'sec3_1 'sec3_2 'ex3_1) .
6 rewrites: 49712 in 32ms cpu (34ms real) (1553500 rewrites/second)
7 result Bool: false
```

Listing 4.5: Aceitação de palavras para o exemplo 3.3

4.3 Transformador de Learn para Teorias de Reescrita

Como alternativa, apresentamos também um transformador para teorias de reescrita. Evidentemente, tudo o que é implementado em Maude é uma teoria de reescrita, contudo esta implementação não tem em vista outro formalismo, como as gramáticas regulares. A expressividade usada na saída deste transformador exige recursos mais avançados.

Este transformador foi implementado antes do apresentado na Seção 4.2 e usado para refinar a compreensão do projeto, contudo até o momento da redação deste texto ainda apresenta importância, pois o transformador para HTML está implementado a partir dos módulos de sistema produzidos por este transformador. Acreditamos que sua semântica é de mais fácil compreensão. No entanto, o transformador descrito na Seção 4.2 é mais eficiente, por manipular somente termos sem variáveis e implementar uma caracterização que nos diz exatamente a qual classe de linguagem Learn pertence, isto é, as linguagens regulares. Saliente-se que são equivalentes, pois produzem resultados isomórficos.

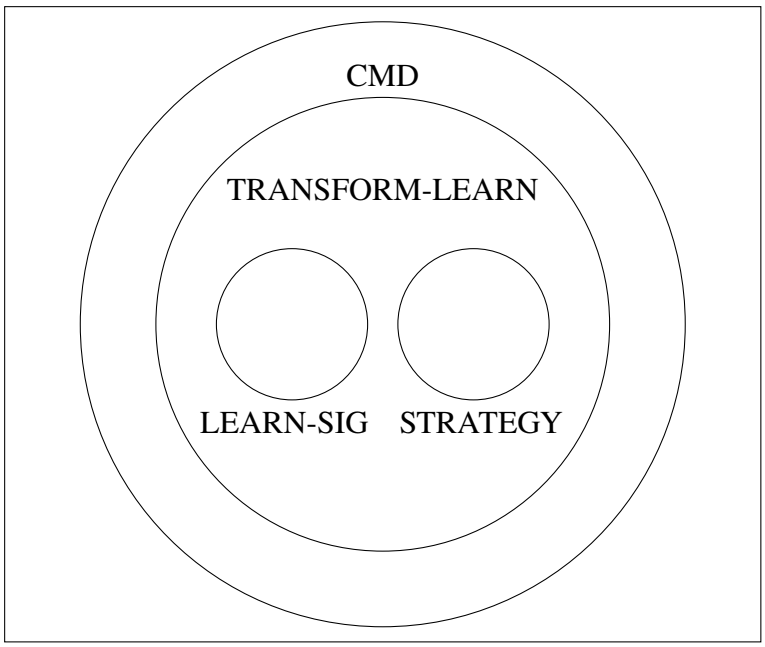


Figura 4.2: Relação de inclusão dos módulos principais do transformador

Seus principais módulos são: (i) LEARN-SIG, com o qual também se faz a análise sintática de Learn; (ii) STRATEGY, que implementa o conjunto de regras responsável pelo cálculo dinâmico dos caminhos de aprendizagem; (iii) TRANSFORM-LEARN, que implementa a meta-função que gera módulos de sistema em Maude a partir de descrições em Learn; e (iv) CMD, que, analogamente ao módulo homônimo descrito na Seção 4.2, encapsula os comandos do transformador numa interface mais simples para o usuário.

Estes módulos são incluídos uns nos outros conforme o diagrama da Figura 4.3. LEARN-SIG e STRATEGY são independentes entre si; podemos entender que o primeiro é a sintaxe de Learn e o segundo sua semântica. Em TRANSFORM-LEARN, que estende a ambos, é descrita sua relação por meio da função de tradução. A seguir exemplificamos algumas das declarações de cada módulo.

O módulo STRATEGY, conforme dito acima, é responsável pelo cálculo dos caminhos de aprendizagem. Isto é implementado através de duas regras de reescrita e levando em consideração a estratégia de ensino. Vale recordar que na ausência de uma estratégia definida, toma-se a estratégia "padrão", ou seja, é assumida a ausência de dependências entre quaisquer objetos de aprendizagem.

```
1  crl state(S, AS, ϵ) ⇒ state(S, (AS − A), A) if choose(AS) ⇒ A ∧ pred(S, A) = ∅ .
2  crl state(S, AS, AL) ⇒ state(S, ((AS − A) B), (AL ; A)) if (choose(AS) ⇒ A) ∧ (B := all − AS) ∧ (pred(S, A) ⊆ AL) ∧ (AL ≠ ϵ) .
```

Nas regras, *S* é o identificador da estratégia, *AS* e *B* são conjuntos de identificadores de objetos de aprendizagem, *AL* é uma lista de identificadores de objetos de aprendizagem e a constante *all* é o conjunto de todos os identificadores de objetos de aprendizagem contidos na especificação do curso. Note que as transições se dão entre estados que são constituídos de estratégia, conjunto de objetos a menos de último e caminho de aprendizado já percorrido.

São elas, portanto, descrições de transições de estados condicionadas pela estratégia de ensino. Um estado é correntemente alcançável se seus predecessores já foram previamente alcançados, ou seja, se seus pré-requisitos foram atendidos. A primeira regra rege a transição a partir do estado inicial, para o qual não se retorna, enquanto a segunda regula todas as demais transições.

É possível olhar para a Figura 3.1 e nela enxergar o comportamento das regras acima. Primeiro o que foi dito sobre o significado de cada regra e também por vermos no estado inicial todos os objetos e nos demais estados todos menos a última escolha. Os estados dados pelas regras seguem esse padrão não apenas para o caso sem estratégia, mas para todo caso e aplicam sobre as transições uma condição, relacionada à ordem dada pela estratégia. Sendo assim, podemos entender que as regras de STRATEGY implementam o autômato do curso sem estratégia e, condicionadas à estratégia, podem adicionar dinamicamente transições previamente "removidas".

Ainda em STRATEGY, também é digno de destaque a operação *init* declarada com o seguinte operador:

```
1  op init : Strategy → State .
```

com uma equação que o identifica a um estado contendo todos os identificadores de objetos de aprendizagem e um caminho de aprendizado vazio, além de inicializar a estratégia, sendo este o estado inicial.

```
1  eq init(S) = state(S, all, ϵ) .
```

A principal declaração do módulo TRANSFORM-LEARN é a função *transformLearnCourse* que, dado um termo da sorte *LearnCourse*, produz um módulo de sistema em Maude, um termo do sorte *SModule*, onde *Cid* é um identificador, *SObj* é um conjunto de identificadores de objetos de aprendizagem, *Lid* é um identificador de um objeto de aprendizagem, CS é um conjunto de estratégias de ensino e *LODecls* é um grupo de declarações de objetos de aprendizagem. Note que o módulo gerado inclui STRATEGY.

```
1  op transformLearnCourse : LearnCourse → SModule .
```

```
eq transformLearnCourse(course on Cid teaches SObj and Lid with CS LOdecls) =
  mod q(Cid) is
    (including 'BOOL . including 'STRATEGY .)
    sorts none .
    none none none
    (eq 'all.LOSet = transformOverSetObj(SObj, Lid) [none] .)
     transformCourseSpecs(CS) transformLearnObjDecls(LOdecls)
    none
  endm .
```

Por fim, o módulo CMD deste transformador declara uma interface para que o usuário possa fazer simulações (animações), análises de alcançabilidade e explorar o funcionamento do curso gerado, bem como usar as funções do transformador de um modo mais direto. Em particular, por exemplo, a função *search* chama a *metaSearch* composta com *getTerm* e *downTerm*, encapsulando assim a meta-representação dos módulos de Learn do usuário.

```
op search : LearnCourse Qid Bound Nat → State .
eq search(LC, Q, B, N) = downTerm(getTerm(metaSearch(transform(LC), 'init['st[upTerm(string(Q))]], 'S:State, nil, '∗, B, N)), st) .
```

Exemplos. Consideremos agora o exemplo da Listing 3.2. Ao carregarmos o transformador no interpretador Maude, executamos o comando na Listing 4.6, onde eg2 é uma constante da sorte LearnCourse que contém a descrição Learn do exemplo. A saída é a meta-representação da descrição.

```
reduce in TEST : transform(eg2) .
rewrites: 507 in 0ms cpu (0ms real) (1701342 rewrites/second)
result SModule: mod 'Formal'Languages is
  including 'BOOL .
  including 'STRATEGY .
  sorts none . none none none
  eq 'all.LOSet = '__['__['lo['"ex3_1".String],'__['lo['"fig3_2".String],'__['lo['"int3".String],'__['lo['"sec2_1".String],'__['lo['"sec2_2".
       String],'__['lo['"sec2_4".String],'__['lo['"sec3_1".String],'__['lo['"sec3_2".String],'__['lo['"sec3_3".String],'lo['"sec3_7".String
       ]]]]]]]]],'lo['"ex3_2".String]] [none] .
  . . .
  ceq 'pred['st['"book".String],'LO:LearningObject] = 'lo['"ex3_1".String] if '_subset_['LO:LearningObject,'lo['"ex3_2".String]] = 'true.
       Bool [none] .
  ceq 'pred['st['"class".String],'LO:LearningObject] = 'lo['"ex3_1".String] if '_subset_['LO:LearningObject,'lo['"ex3_2".String]] = 'true.
       Bool [none] .
  . . .
  none
endm
```

Listing 4.6: Meta-representação da descrição Learn do exemplo 3.2

Sobre o mesmo exemplo aplicamos o comando *search*, do módulo CMD, e mostramos na Listing 4.7 o resultado, o estado alcançado no curso em questão com a estratégia *class*, que traz no terceiro argumento o caminho seguido de int3 até fig3_2. Compare com a execução na Listing 4.4.

```
reduce in TEST : search(eg2, 'class, 4, 7) .
rewrites: 572 in 0ms cpu (0ms real) (1692307 rewrites/second)
result State: state(st("class"), lo("ex3_1") lo("ex3_2") lo("int3") lo("sec2_1") lo("sec2_2") lo("sec2_4") lo("sec3_1") lo("sec3_2") lo("sec3_3
      ") lo("sec3_7"), lo("int3") ; lo("sec3_1") ; lo("sec3_2") ; lo("fig3_2"))
```

Listing 4.7: Animação de descrição Learn com search, para o exemplo 3.2

Nas Listings 3.3 e 4.9 aplicamos os mesmos comandos sobre a descrição Learn do exemplo 3.3, onde eg3 é também uma constante que contém a descrição supracitada. Como resultado temos uma visão de sua meta-representação, que, como dito na Seção 3.1, traz maior generalidade. Em seguida vemos também o estado alcançado pelo *search* e o caminho de aprendizagem percorrido.

```
reduce in TEST : transform(eg3) .
rewrites: 143 in 0ms cpu (0ms real) (~ rewrites/second)
result SModule: mod 'Linguagens'Formais is
  including 'BOOL .
  including 'STRATEGY .
  sorts none . none none none
  eq 'all.LOSet = '__['__['lo['"ex3_1".String],'__['lo['"fig3_2".String],'__['lo['"int3".String],'__['lo['"sec3_1".String],'lo['"sec3_2".String
      ]]]]],'lo['"ex3_2".String]] [none] .
  eq 'exercises.LOSet = '__['lo['"ex3_1".String],'lo['"ex3_2".String]] [none] .
  . . .
  ceq 'pred['st['"ex".String],'LO:LearningObject] = '_-_['all.LOSet,'exercises.LOSet] if '_subset_['LO:LearningObject,'exercises.LOSet]
      = 'true.Bool [none] .
  none
endm
```

Listing 4.8: Meta-representação da descrição Learn do exemplo 3.3

```
reduce in TEST : search(eg3, 'ex, 4, 7) .
rewrites: 249 in 0ms cpu (7ms real) (~ rewrites/second)
result State: state(st("ex"), lo("ex3_1") lo("ex3_2") lo("fig3_2") lo("int3")
    lo("sec3_1"), lo("fig3_2") ; lo("sec3_2"))
```

Listing 4.9: Animação de descrição Learn com search, para o exemplo 3.3

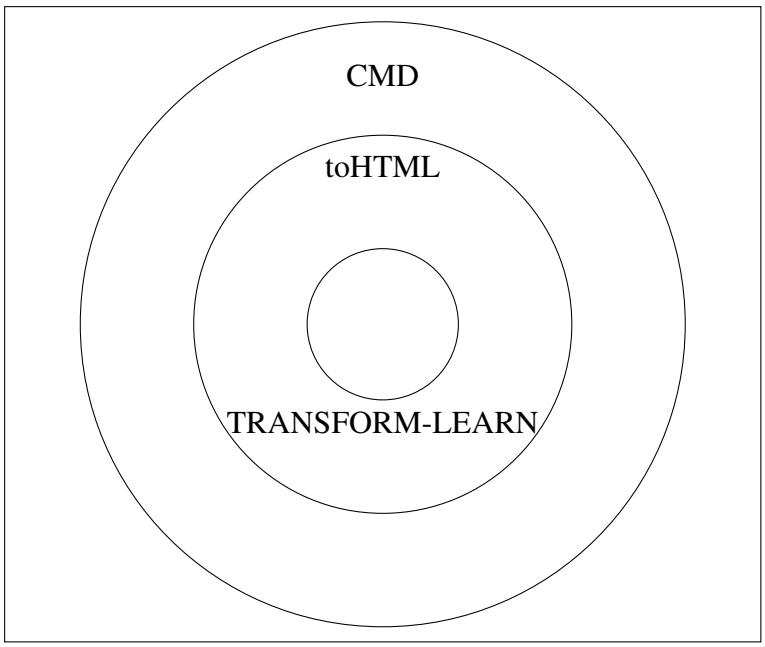


Figura 4.3: Relação de inclusão do módulo toHTML

4.4 Transformador de Teorias de Reescrita Learn para HTML

Até onde sabemos os MOOCs não suportam algo como a semântica das estratégias de Learn e não dão meios claros de integrar novas ferramentas, sobretudo não-gráficas, para construir cursos em suas plataformas. Diante disso ainda há muito a ser feito para que Learn atenda ao fim que se propõe.

Todavia, como prova de conceito, a fim de mostrar uma implementação executável das ideias descritas neste trabalho em um ambiente que simule a experiência do aluno, este transformador, LearnHTML, implementa ainda um módulo, toHTML, para gerar uma página HTML (com elementos em Javascript) no modelo *single-page application* a partir da semântica do curso descrita como teorias de reescrita em Maude. Na Figura 4.4 ilustramos a relação de inclusão do módulo toHTML com os demais do transformador. Ele inclui o TRANSFORM-LEARN e é incluído pelo CMD.

O operador *convertHTML* é o responsável por transformar um módulo de sistema com a semântica do curso em uma String que, por simplicidade, é o formato de saída para o código-fonte HTML externado pelo módulo, de modo que pode ser salvo num arquivo texto.

```
1 op convertHTML : SModule → String .
2 eq html(LC) = convertHTML(transform(LC)) .
```

E o módulo CMD provê uma função que concatena *transformLearnCourse* e *convertHTML*

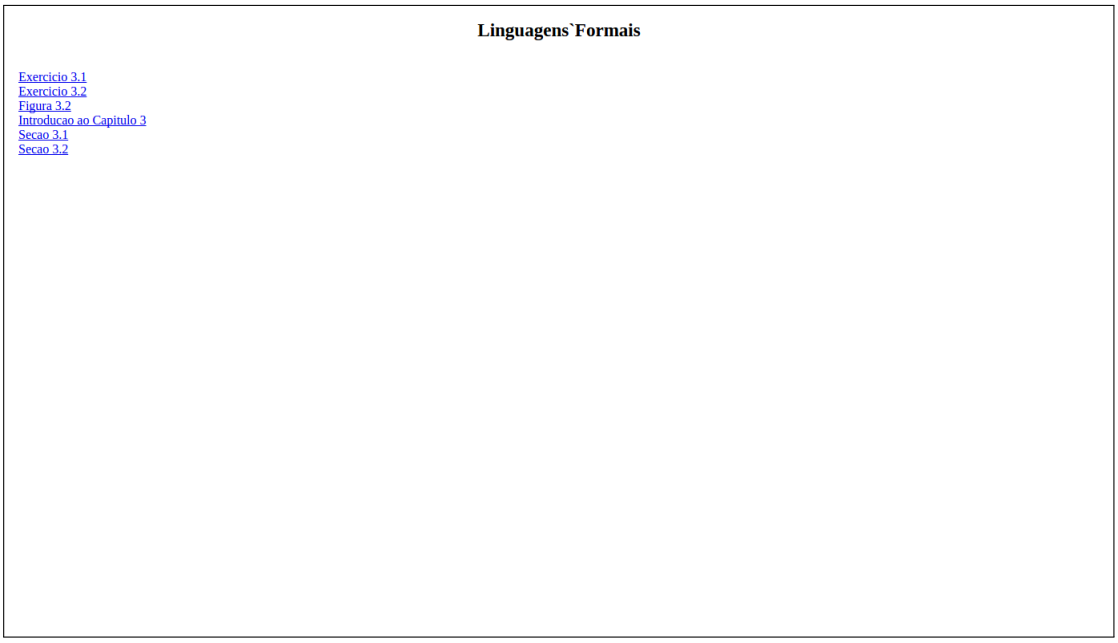


Figura 4.4: Página inicial do exemplo 3.1

para gerar a página diretamente da especificação em Learn.

```
1  op html : LearnCourse → String .
```

O Learn Maude Toolkit implementa também um script para automatizar a compilação de um arquivo de código-fonte em Learn para um arquivo em HTML, fazendo uso do operador em CMD.

Através da compilação das descrições apresentadas na Seção 3.1 obtemos os exemplos a seguir. Saliente-se que a formatação vista nas Figuras é construída nos exemplos através da inserção de rótulos HTML diretamente no código Learn. A Figura 4.4 diz respeito ao estado inicial do exemplo da Listing 3.1. Como é o caso sem estratégia vemos que desde o começo todos os objetos de aprendizagem declarados no curso estão inicialmente disponíveis. Diferentemente, na Figura 4.7, relativa ao exemplo contido na Listing 3.3, os exercícios não estão inicialmente disponíveis, mas estarão tão logo sejam estudados os demais objetos de aprendizagem, que é o caso da Figura 4.8. Comparando estas duas últimas podemos notar a ausência do link para o objeto corrente e visualizar nelas um nó sendo mostrado. E sobre o exemplo da Listing 3.2 apresentamos a mesma seção mostrada nas figuras anteriores vista neste caso de acordo com cada estratégia, na Figura 4.5 segundo a estratégia *book* e na Figura 4.6 segundo a estratégia *class*. Vemos os links para os objetos já percorridos e para aquele liberado após o acesso da seção corrente.

Ainda, conforme dissemos, o arquivo gerado pode ser incluído em um curso no Moo-

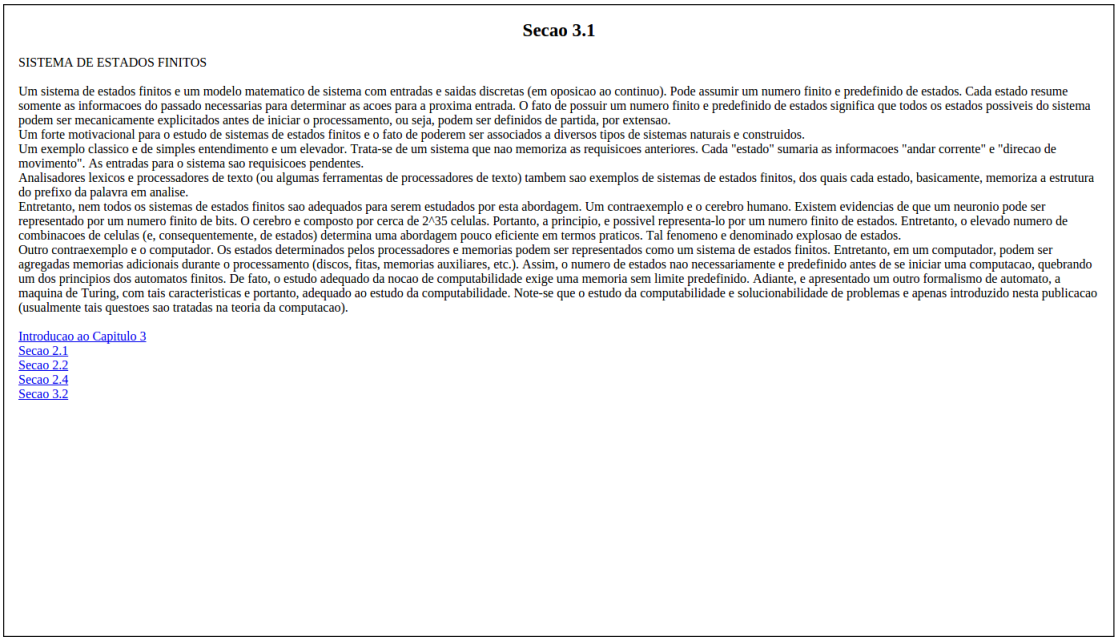


Figura 4.5: Seção 3.1 do exemplo 3.2, na estratégia *book*

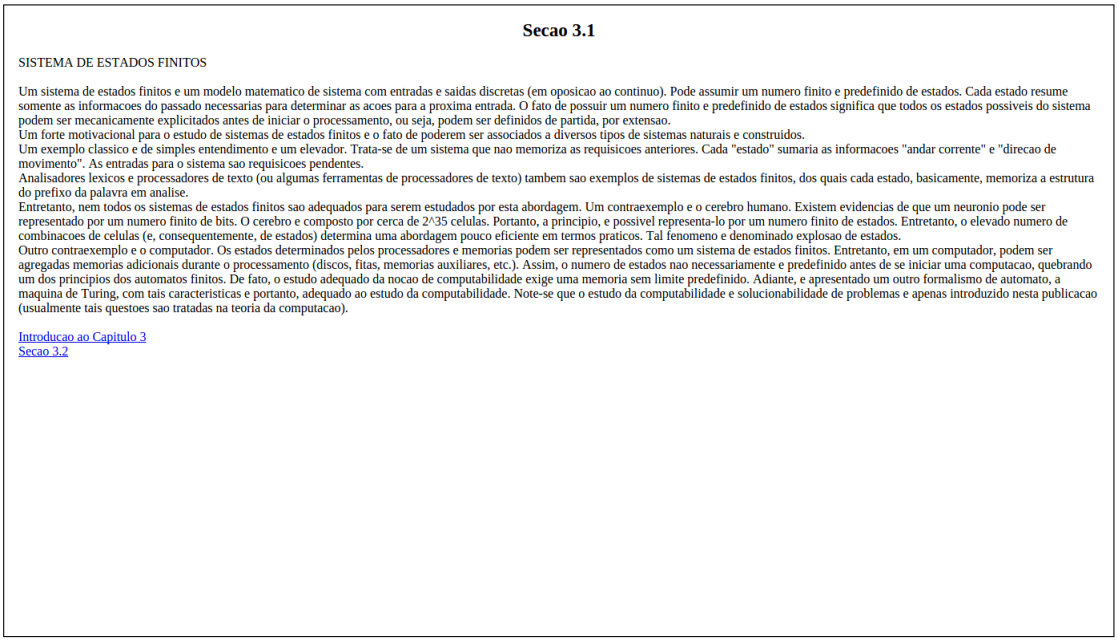
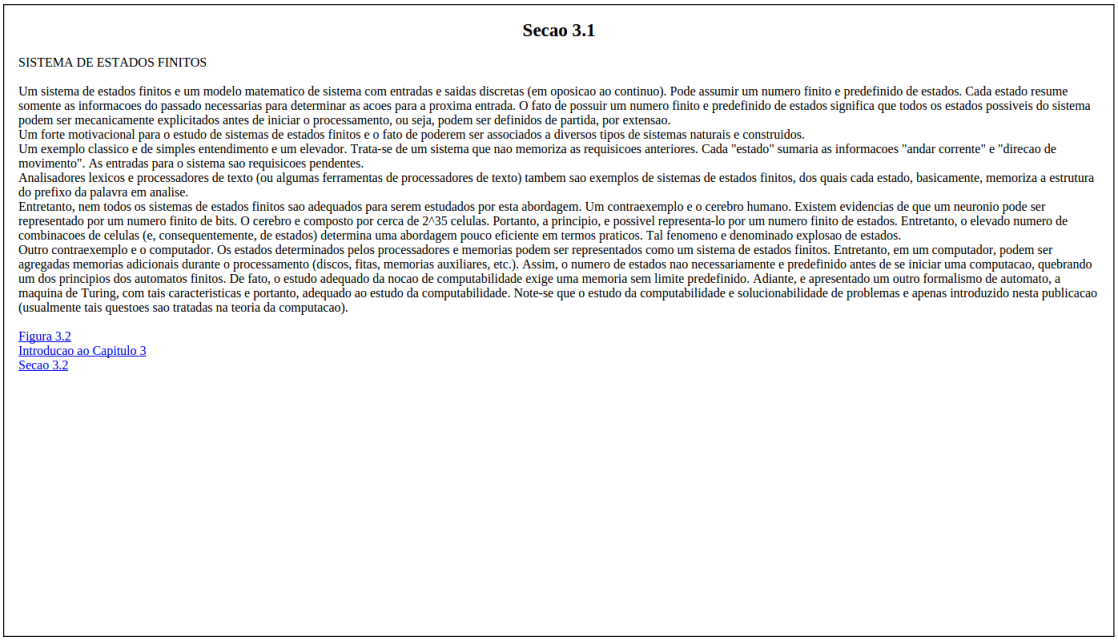


Figura 4.6: Seção 3.1 do exemplo 3.2, na estratégia *class*



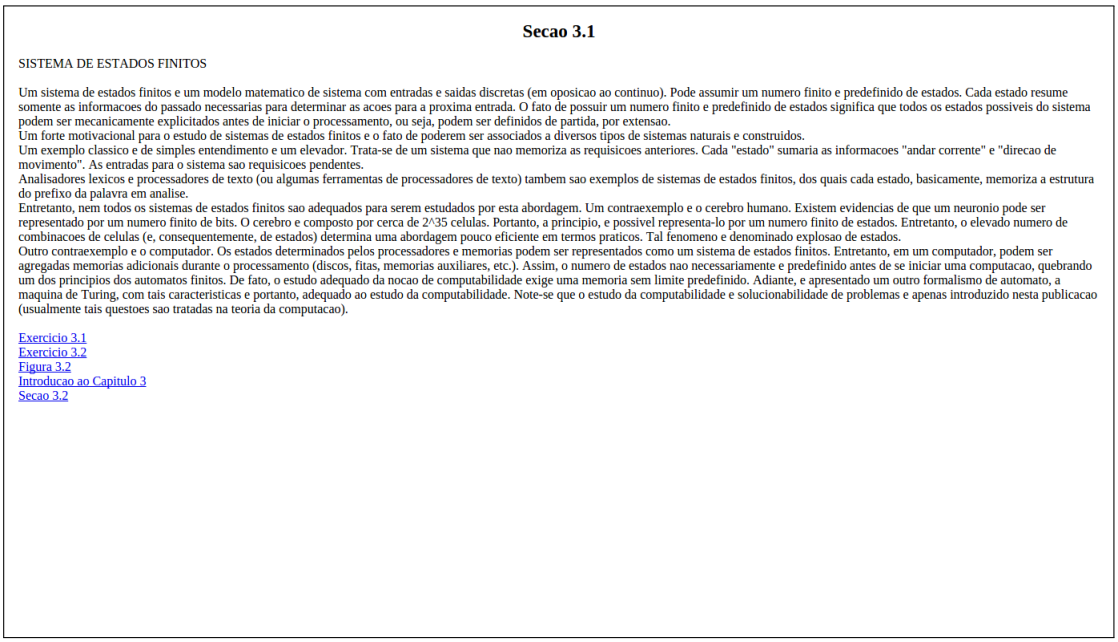
Secao 3.1

SISTEMA DE ESTADOS FINITOS

Um sistema de estados finitos e um modelo matematico de sistema com entradas e saidas discretas (em oposicao ao continuo). Pode assumir um numero finito e predefinido de estados. Cada estado resume somente as informacoes do passado necessarias para determinar as acoes para a proxima entrada. O fato de possuir um numero finito e predefinido de estados significa que todos os estados possiveis do sistema podem ser mecanicamente explicitados antes de iniciar o processamento, ou seja, podem ser definidos de partida, por extensao.
Um forte motivacional para o estudo de sistemas de estados finitos e o fato de poderem ser associados a diversos tipos de sistemas naturais e construidos.
Um exemplo classico e de simples entendimento e um elevador. Trata-se de um sistema que nao memoriza as requisicoes anteriores. Cada "estado" sumaria as informacoes "andar corrente" e "direcao de movimento". As entradas para o sistema sao requisicoes pendentes.
Analisadores lexicos e processadores de texto (ou algumas ferramentas de processadores de texto) tambem sao exemplos de sistemas de estados finitos, dos quais cada estado, basicamente, memoriza a estrutura do prefixo da palavra em analise.
Entretanto, nem todos os sistemas de estados finitos sao adequados para serem estudados por esta abordagem. Um contraexemplo e o cerebro humano. Existem evidencias de que um neuronio pode ser representado por um numero finito de bits. O cerebro e composto por cerca de 2^35 celulas. Portanto, a principio, e possivel representa-lo por um numero finito de estados. Entretanto, o elevado numero de combinacoes de celulas (e, consequentemente, de estados) determina uma abordagem pouco eficiente em termos praticos. Tal fenomeno e denominado explosao de estados.
Outro contraexemplo e o computador. Os estados determinados pelos processadores e memorias podem ser representados como um sistema de estados finitos. Entretanto, em um computador, podem ser agregadas memorias adicionais durante o processamento (discos, fitas, memorias auxiliares, etc.). Assim, o numero de estados nao necessariamente e predefinido antes de se iniciar uma computacao, quebrando um dos principios dos automatos finitos. De fato, o estudo adequado da nocao de computabilidade exige uma memoria sem limite predefinido. Adiante, e apresentado um outro formalismo de automato, a maquina de Turing, com tais caracteristicas e portanto, adequado ao estudo da computabilidade. Note-se que o estudo da computabilidade e solucionabilidade de problemas e apenas introduzido nesta publicacao (usualmente tais questoes sao tratadas na teoria da computacao).

Figura 3.2
Introducao ao Capitulo 3
Secao 3.2

Figura 4.7: Seção 3.1 do exemplo 3.3



Secao 3.1

SISTEMA DE ESTADOS FINITOS

Um sistema de estados finitos e um modelo matematico de sistema com entradas e saidas discretas (em oposicao ao continuo). Pode assumir um numero finito e predefinido de estados. Cada estado resume somente as informacoes do passado necessarias para determinar as acoes para a proxima entrada. O fato de possuir um numero finito e predefinido de estados significa que todos os estados possiveis do sistema podem ser mecanicamente explicitados antes de iniciar o processamento, ou seja, podem ser definidos de partida, por extensao.
Um forte motivacional para o estudo de sistemas de estados finitos e o fato de poderem ser associados a diversos tipos de sistemas naturais e construidos.
Um exemplo classico e de simples entendimento e um elevador. Trata-se de um sistema que nao memoriza as requisicoes anteriores. Cada "estado" sumaria as informacoes "andar corrente" e "direcao de movimento". As entradas para o sistema sao requisicoes pendentes.
Analisadores lexicos e processadores de texto (ou algumas ferramentas de processadores de texto) tambem sao exemplos de sistemas de estados finitos, dos quais cada estado, basicamente, memoriza a estrutura do prefixo da palavra em analise.
Entretanto, nem todos os sistemas de estados finitos sao adequados para serem estudados por esta abordagem. Um contraexemplo e o cerebro humano. Existem evidencias de que um neuronio pode ser representado por um numero finito de bits. O cerebro e composto por cerca de 2^35 celulas. Portanto, a principio, e possivel representa-lo por um numero finito de estados. Entretanto, o elevado numero de combinacoes de celulas (e, consequentemente, de estados) determina uma abordagem pouco eficiente em termos praticos. Tal fenomeno e denominado explosao de estados.
Outro contraexemplo e o computador. Os estados determinados pelos processadores e memorias podem ser representados como um sistema de estados finitos. Entretanto, em um computador, podem ser agregadas memorias adicionais durante o processamento (discos, fitas, memorias auxiliares, etc.). Assim, o numero de estados nao necessariamente e predefinido antes de se iniciar uma computacao, quebrando um dos principios dos automatos finitos. De fato, o estudo adequado da nocao de computabilidade exige uma memoria sem limite predefinido. Adiante, e apresentado um outro formalismo de automato, a maquina de Turing, com tais caracteristicas e portanto, adequado ao estudo da computabilidade. Note-se que o estudo da computabilidade e solucionabilidade de problemas e apenas introduzido nesta publicacao (usualmente tais questoes sao tratadas na teoria da computacao).

Exercicio 3.1
Exercicio 3.2
Figura 3.2
Introducao ao Capitulo 3
Secao 3.2

Figura 4.8: Seção 3.1 do exemplo 3.3, após ter percorrido os objetos iniciais

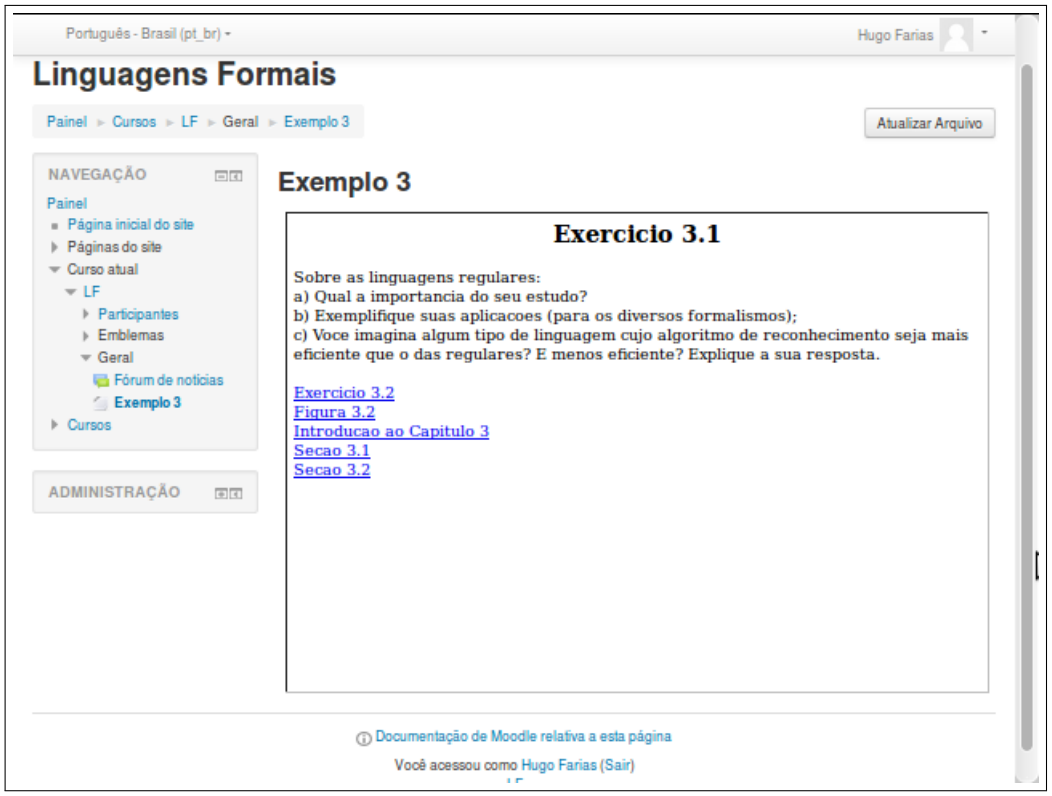


Figura 4.9: Execução do HTML do curso no Moodle

dle e poderá ser executado dentro dele, conforme a Figura 4.9. Possivelmente uma extensão à plataforma pudesse tornar o curso gerado em Maude mais integrado ao ambiente.

Capítulo 5

Exemplos

Na Seção 4.4 apresentamos uma prévia do objetivo para Learn, gerar cursos online. Todavia o foco deste trabalho é a definição formal da linguagem. A partir desta poderá se desenvolver uma correta conversão para as plataformas de ensino disponíveis.

Diante disto, neste capítulo desenvolvemos um exemplo da construção de um curso demonstrando a expressividade de Learn e o efeito sobre a semântica da aplicação de uma estratégia. Para tal faremos uso da ferramenta apresentada na Seção 4.2.

Usaremos uma adaptação do exemplo da Listing 4.1. Nosso ponto de partida será um monoide livre tendo o alfabeto do exemplo como conjunto gerador.

O monoide reconhece qualquer palavra formada por símbolos do alfabeto. Um curso sem estratégia é visto como uma restrição sobre um monoide onde são rejeitadas as palavras que tenham como sub-palavra um mesmo símbolo contíguo, ou seja, o autômato relacionado a um curso Learn não reconhece uma palavra onde a escolha depois de um objeto seja ele mesmo.

Nosso curso sem estratégia é descrito na Listing 5.1. O autômato que reconhece seus caminhos de aprendizagem é representado na Figura 5.2. Note que é o mesmo autômato da Figura 3.1, apenas com rótulos diferentes nos estados, a fim de evidenciar a relação com as gramáticas regulares apresentadas na Seção 3.2.

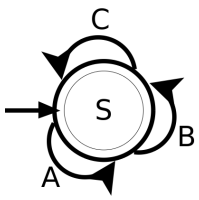


Figura 5.1: Monoide Livre

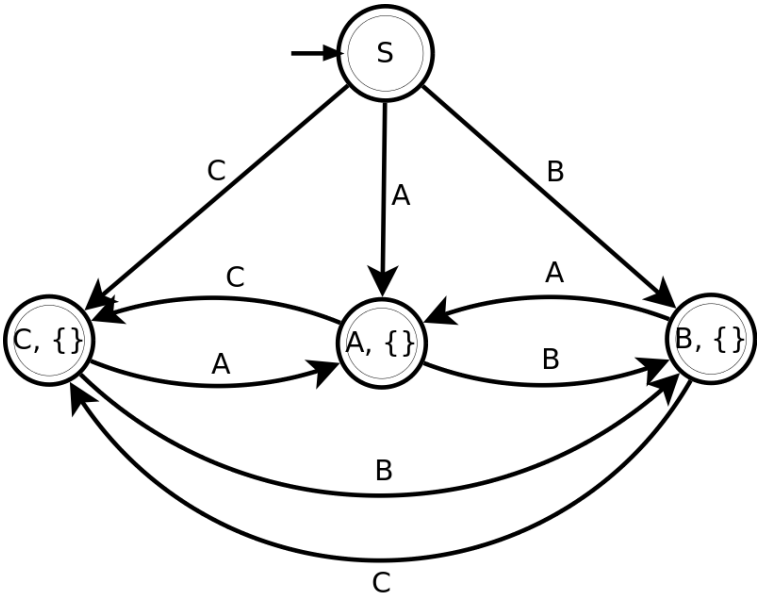


Figura 5.2: Autômato para o caso sem estratégia

```
course on "TEST"
   teaches < 'A >, < 'B > and < 'C >
learning object < 'A > has title "TEST A"
learning object < 'B > has title "TEST B"
learning object < 'C > has title "TEST C"
```

Listing 5.1: Curso sem estratégia

Conforme mencionado previamente, a imposição de uma restrição sobre um autômato aumenta o número de seus estados e transições e insere complexidade.

Também por simplicidade, definimos um módulo Maude para relacionar as descrições a constantes e aplicamos a função getGrammar à constante noStrategy, que contém a especificação do curso sem estratégia.

```
reduce in EXAMPLES : getGrammar(noStrategy, 'default) .
rewrites: 389 in 0ms cpu (0ms real) (~ rewrites/second)
result ProductionSet: (S → 'A)
(S → 'B)
(S → 'C)
(S → epsilon)
(S → 'A ['A,{}])
(S → 'B ['B,{}])
(S → 'C ['C,{}])
(['A,{}] → 'B)
(['A,{}] → 'C)
(['A,{}] → 'B ['B,{}])
(['A,{}] → 'C ['C,{}])
```

```
14 (['B,{}] → 'A)
15 (['B,{}] → 'C)
16 (['B,{}] → 'A ['A,{}])
17 (['B,{}] → 'C ['C,{}])
18 (['C,{}] → 'A)
19 (['C,{}] → 'B)
20 (['C,{}] → 'A ['A,{}])
21 ['C,{}] → 'B ['B,{}]
```

Listing 5.2: Geração da gramática para o caso sem estratégia

Em seguida, acrescemos o curso de uma estratégia simples: a restrição de que o objeto C só pode ser estudado uma vez que o B o tenha sido, B $\leq$ C; descrita na Listing 4.1. O mesmo exemplo dado nas Seções 3.2 e 4.2. Na Figura 5.3 temos o autômato para este caso. Podemos ver nele com maior clareza o aumento da complexidade, se comparado ao do caso sem estratégia.

A geração da gramática do curso é mostrada novamente na Listing 5.3.

```
1  reduce in EXAMPLES : getGrammar(simpleStrategy, 's) .
2  rewrites: 583 in 0ms cpu (1ms real) (~ rewrites/second)
3  result ProductionSet: (S → 'A)
4  (S → 'B)
5  (S → epsilon)
6  (S → 'A ['A,{}])
7  (S → 'B ['B,{'B}])
8  (['A,{}] → 'B)
9  (['A,{}] → 'B ['B,{'B}])
10 (['A,{'B}] → 'B)
11 (['A,{'B}] → 'C)
12 (['A,{'B}] → 'B ['B,{'B}])
13 (['A,{'B}] → 'C ['C,{'B}])
14 (['B,{'B}] → 'A)
15 (['B,{'B}] → 'C)
16 (['B,{'B}] → 'A ['A,{'B}])
17 (['B,{'B}] → 'C ['C,{'B}])
18 (['C,{'B}] → 'A)
19 (['C,{'B}] → 'B)
20 (['C,{'B}] → 'A ['A,{'B}])
21 ['C,{'B}] → 'B ['B,{'B}]
```

Listing 5.3: Geração da gramática para o caso com estratégia simples

E por fim aumentamos a complexidade da estratégia, acrescentando a regra A $\leq$ C e chegando ao curso descrito na Listing 5.4, cujo autômato pode ser visto na Figura 5.4.

```
1 course on "TEST"
2   teaches < 'A >, < 'B > and < 'C >
3 with
4   teaching strategy < 's >
```

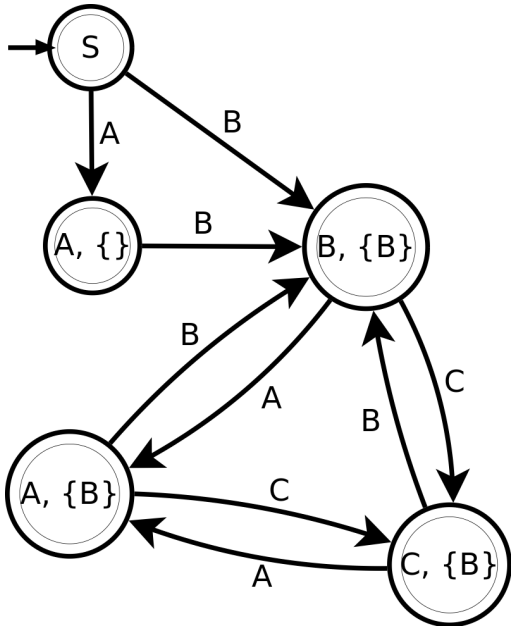


Figura 5.3: Autômato para o caso com estratégia simples

```
    < 'B > before < 'C >,
    < 'A > before < 'C >
learning object < 'A > has title "TEST A"
learning object < 'B > has title "TEST B"
learning object < 'C > has title "TEST C"
```

Listing 5.4: Curso com estratégia mais complexa

A gramática é gerada na Listing 5.5.

```
reduce in EXAMPLES : getGrammar(complexStrategy, 's) .
rewrites: 1242 in 0ms cpu (1ms real) (~ rewrites/second)
result ProductionSet: (S → 'A)
(S → 'B)
(S → epsilon)
(S → 'A ['A,{'A}])
(S → 'B ['B,{'B}])
(['A,{'A}] → 'B)
(['A,{'A}] → 'B ['B,{'A, 'B}])
(['A,{'A, 'B}] → 'B)
(['A,{'A, 'B}] → 'C)
(['A,{'A, 'B}] → 'B ['B,{'A, 'B}])
(['A,{'A, 'B}] → 'C ['C,{'A, 'B}])
(['B,{'B}] → 'A)
(['B,{'B}] → 'A ['A,{'A, 'B}])
(['B,{'A, 'B}] → 'A)
(['B,{'A, 'B}] → 'C)
(['B,{'A, 'B}] → 'A ['A,{'A, 'B}])
(['B,{'A, 'B}] → 'C ['C,{'A, 'B}])
(['C,{'A, 'B}] → 'A)
```

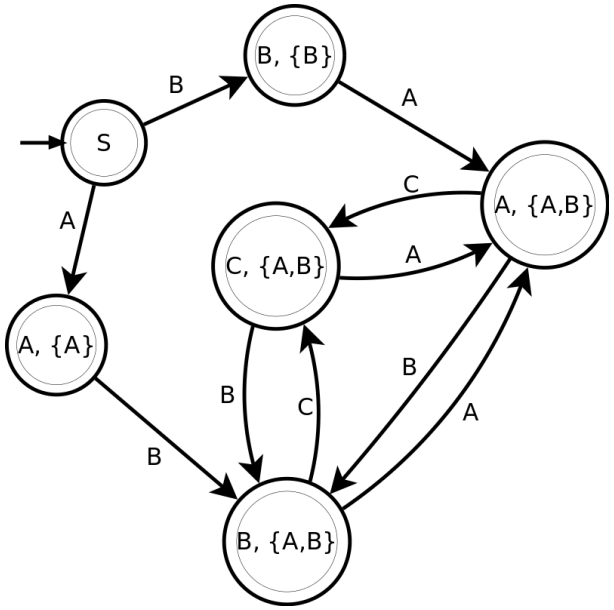


Figura 5.4: Autômato para o caso com estratégia mais complexa

```
21 (['C,{'A, 'B}] → 'B)
22 (['C,{'A, 'B}] → 'A ['A,{'A, 'B}])
23 ['C,{'A, 'B}] → 'B ['B,{'A, 'B}]
```

Listing 5.5: Geração da gramática para o caso com estratégia mais complexa

Frise-se a proporcionalidade do aumento da complexidade à restrição imposta pela estratégia, o que é mais nítido nas figuras dos autômatos, mas também perceptível no comportamento descrito pelas gramáticas.

Capítulo 6

Conclusão e trabalhos futuros

Propomos a linguagem Learn para a declaração de cursos, os quais têm a semântica de gramáticas regulares que constroem os possíveis caminhos de aprendizagem. Uma produção indica a escolha de um objeto para o estudo, dentre as possibilidades. Sendo assim, uma computação representa um caminho que pode-se tomar para estudar um dado conjunto de objetos de aprendizagem. Contribuímos com a linguagem, sua semântica formal e um protótipo de ambiente de execução escrito na linguagem Maude, onde computações são identificadas com reescritas em um sistema adequado de reescrita.

Concluímos tendo apresentado uma sintaxe e um modelo teórico para a fundamentação de Learn, em vista de seu futuro desenvolvimento sobre as ideias aqui estabelecidas a fim de alcançar o objetivo dado de princípio.

Dentre os trabalhos futuros contamos, junto ao aperfeiçoamento da implementação, uma eventual relação entre Learn e Nautilus [18], uma linguagem de especificação de sistemas de objetos concorrentes associada aos autômatos não-sequenciais e, claro, a compatibilização de Learn com as plataformas MOOC.

Referências Bibliográficas

[1] xConsortium, "edX on line courses portal". `http://www.edx.org`.

[2] D. Koller e A. Ng, "Coursera online courses portal". `http://coursera.org`.

[3] "Udacity". `https://www.udacity.com/`.

[4] M. Dougiamas, "Moodle learning platform". `https://moodle.org`.

[5] "TelEduc". `http://www.teleduc.org.br/`.

[6] R. L. de Souza Dutra e L. M. R. Tarouco, "Objetos de aprendizagem: Uma comparação entre SCORM e IMS Learning Design", *RENOTE*, vol. 4, no. 1, 2006.

[7] G. Lütolf, "Zugänglichkeit von geographischen e-learning-kursen für sehbehinderte und blinde am beispiel von gitta", Tese de Mestrado, University of Zurich, 2006.

[8] "Duolingo". `https://www.duolingo.com/`.

[9] P. B. Menezes, *Linguagens Formais e Autômatos*, vol. 3 of *Série livros didáticos*. Bookman, 6 ed., 2011.

[10] P. B. Menezes e J. P. Machado, "Web courses are automata: a categorial framework", em *Proceedings of 2nd Workshop of Formal Methods*, pág. 79–88, 1999.

[11] P. B. Menezes, A. S. C. Sernadas e J. F. Costa, "Nonsequential automata semantics for concurrent, object-based language", vol. 14 of *ENTCS*, pág. 245–273, 1997.

[12] P. B. Menezes e E. H. Haeusler, *Teoria das Categorias para Ciência da Computação*, vol. 12 of *Série livros didáticos*. Bookman, 2 ed., 2008.

[13] C. Braga e B. Lopes, "Towards reasoning in dynamic logics with rewriting logic: the petri-pdl case", 2015.

[14] A. Bouhoula, J.-P. Jouannaud e J. Meseguer, “Specification and proof in membership equational logic”, *Theor. Comput. Sci.*, vol. 236, pág. 35–132, abril de 2000.

[15] J. Meseguer, “Twenty years of rewriting logic”, *The Journal of Logic and Algebraic Programming*, vol. 81, no. 7, pág. 721–781, 2012.

[16] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer e C. Talcott, *All about maude-a high-performance logical framework: how to specify, program and verify systems in rewriting logic*. Springer-Verlag, 2007.

[17] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martı-Oliet, J. Meseguer e C. Talcott, “Maude manual (version 2.6)”, *University of Illinois, Urbana-Champaign*, vol. 1, no. 3, pág. 4–6, 2011.

[18] C. N. Fuzitaki, P. B. Menezes, J. P. Machado e F. D’Andrea, “Nautilus, a concurrent diagrammatic specification and programming language”, *The Journal of Supercomputing*, vol. 36, no. 1, pág. 51–81, 2006.

[19] P. R. Polsani, “Use and abuse of reusable learning objects”, *Journal of Digital information*, vol. 3, no. 4, 2006.

Apêndice A

Objetos de Aprendizagem

A locução "Objeto de Aprendizagem"(do inglês, *Learning Object*) não é criação deste trabalho, mas antes um conceito da educação assistida por computador estudado e pesquisado na chamada Teoria de Objetos de Aprendizagem (*Learning Objects Theory*).

Por conveniência, Learn faz uso deste conceito em sua composição (também eLML o faz), a fim de aproveitar algumas das ideias relacionadas a ele, em particular a reusabilidade. Todavia há toda uma discussão na literatura sobre o assunto e tentativas muitas de definição [19].

Dentre elas destacamos uma das mais utilizadas, apesar de ampla. De acordo com o *Learning Objects Metadata Workgroup*, um objeto de aprendizagem é "qualquer entidade, digital ou não digital, que possa ser utilizada, reutilizada ou referenciada durante o aprendizado suportado por tecnologias".

Apêndice B

Código

B.1 Analisador Sintático

```
fmod LEARN-SIG is
        pr QID . pr STRING .
        sorts SetObj CourseDecl ObjSpecs LearnCourse CourseID CourseSpecs ExDecl StrID SetStr
            TeachStrDecl StrDecl LearnID StrExp LearnObjDecl SetLOAttrib ObjAttrib
            LearnObjDecls .
        subsort LearnID < SetObj .
        subsort String < CourseID .
        subsort ExDecl TeachStrDecl < CourseSpecs .
        subsort StrDecl < SetStr .
        subsort SetObj < StrExp .
        subsort ObjAttrib < SetLOAttrib .
        subsort LearnObjDecl < LearnObjDecls .

        op <_> : Qid -> LearnID .

        -- Course declarations
        op _,_ : SetObj SetObj -> SetObj [prec 10 assoc comm id: none] .
        op _ _ : CourseDecl LearnObjDecls -> LearnCourse [prec 70] .
        op course on_teaches_ : CourseID LearnID -> CourseDecl [prec 50] .
        op course on_teaches_and_ : CourseID SetObj LearnID -> CourseDecl [prec 50] .
        op course on_teaches_with_ : CourseID LearnID CourseSpecs -> CourseDecl [prec 50] .
        op course on_teaches_and_with_ : CourseID SetObj LearnID CourseSpecs -> CourseDecl [
            prec 50] .
        op none : -> LearnID .
        op none : -> CourseSpecs .
        op _ _ : CourseSpecs CourseSpecs -> CourseSpecs [prec 30 assoc comm id: none] .

        -- Exercise declarations
        op exercise_ : LearnID -> ExDecl .
        op exercises_and_ : SetObj LearnID -> ExDecl [prec 20] .
```

```
        --- Strategy declarations
        op teaching strategy_ _ : LearnID SetStr -> TeachStrDecl [prec 20] .
        op _,_ : SetStr SetStr -> SetStr [prec 10 assoc comm] .
        op _before_ : StrExp StrExp -> StrDecl [prec 5] .
        op _after_ : StrExp StrExp -> StrDecl [prec 5] .
        ops exercises the`exercises all : -> StrExp .
        op _-_ : StrExp StrExp -> StrExp [prec 3] .
        op _but_ : StrExp StrExp -> StrExp [prec 3] .

        --- Learning object declarations
        op learning object_has title_ : LearnID String -> LearnObjDecl [prec 50] .
        op learning object_has title__ : LearnID String SetLOAttrib -> LearnObjDecl [prec 50]
            .
        op text_ : String -> ObjAttrib [prec 20] .
        op image_ : String -> ObjAttrib [prec 20] .
        op __ : SetLOAttrib SetLOAttrib -> SetLOAttrib [prec 30 assoc comm] .
        op __ : LearnObjDecls LearnObjDecls -> LearnObjDecls [prec 60 assoc comm] .

        var S : CourseID . vars O O1 O2 : LearnID . var LO : SetObj . vars SE1 SE2 : StrExp .
            var CS : CourseSpecs .

        eq course on S teaches O = course on S teaches O and none with none .
        eq course on S teaches LO and O = course on S teaches LO and O with none .
        eq course on S teaches O with CS = course on S teaches O and none with CS .

        eq the exercises = exercises .
        eq SE1 but SE2 = SE1 - SE2 .
endfm
```

B.2 Transformador de Learn para Gramáticas

```
fmod GRAMMAR is
        inc QID .
        inc SET*{Qid} .
        sorts V Word Initial T Strategy .
        subsort Initial < V < Word .
        subsort Qid < T < Word .
        subsort Qid < Strategy .
        op epsilon : -> Word .
        op S- : Strategy -> Initial .
        op __ : Word Word -> Word [assoc id: epsilon] .
        op [_:_:_] : T Set{Qid} Strategy -> V .
        op tam : Word -> Nat .
        eq tam(epsilon) = 0 .
        eq tam(Tmn:T W:Word) = 1 + tam(W:Word) .
        eq tam(Vb:V W:Word) = 1 + tam(W:Word) .
endfm
```

```
fmod IM-GRAMMAR is
        inc GRAMMAR .
        sort Sort .
        subsort Qid < Sort .
endfm

fmod PARTIAL-ORDER is
        inc QID .
        sort PartialOrder .
        op `(_,_`) : Qid Qid -> PartialOrder .
endfm

view PartialOrder from TRIV to PARTIAL-ORDER is
        sort Elt to PartialOrder .
endv

mod GRAMMAR_TRANSFORM is
        inc NAT .
        inc META-LEVEL * (op insert to EXinsert, op delete to EXdelete, op _in_ to _EXin_, op
            |_| to EX|_|, op $card to EX$card, op union to EXunion, op intersection to
            EXintersection, op $intersect to EX$intersect, op _\_ to _EX\_, op $diff to
            EX$diff, op _subset_ to _EXsubset_, op _psubset_ to _EXpsubset_, op _`,_ to _>_,
            op __ to _§_, op _;_ to _~_) .
        inc LEARN-SIG .
        inc SET*{Qid} . ——
        inc SET*{PartialOrder} .

        var Cid : CourseID . var SObj ExObj : SetObj . vars Lid : LearnID . var LOdecls :
            LearnObjDecls . vars CS CS' : CourseSpecs . var SStr : SetStr . var StSet : Set{
            PartialOrder} . vars LO StrID Q Q1 Q2 : Qid . vars exp1 exp2 : StrExp . var SQ
            Choices : Set{Qid} .

        op qid : LearnID -> Qid .
        eq qid(< Q:Qid >) = Q:Qid .

        op set : SetObj -> Set{Qid} .
        eq set(none) = {} .
        eq set(Lid, SObj) = union(set(SObj), {qid(Lid)}) .

        op strSet : SetStr SetObj SetObj -> Set{PartialOrder} .
        eq strSet(exp1 before exp2, SObj, ExObj) = cartesian(redStr(exp1, SObj, ExObj), redStr
            (exp2, SObj, ExObj)) .
        eq strSet(exp1 after exp2, SObj, ExObj) = cartesian(redStr(exp2, SObj, ExObj), redStr(
            exp1, SObj, ExObj)) .
        eq strSet((exp1 before exp2, SStr), SObj, ExObj) = union(strSet(SStr, SObj, ExObj),
            cartesian(redStr(exp1, SObj, ExObj), redStr(exp2, SObj, ExObj))) .
        eq strSet((exp1 after exp2, SStr), SObj, ExObj) = union(strSet(SStr, SObj, ExObj),
            cartesian(redStr(exp2, SObj, ExObj), redStr(exp1, SObj, ExObj))) .
```

```
op redStr : StrExp SetObj SetObj -> Set{Qid} .
eq redStr(all, SObj, ExObj) = set(SObj) .
eq redStr(exercises, SObj, ExObj) = set(ExObj) .
eq redStr(Lid, SObj, ExObj) = {qid(Lid)} .
eq redStr(exp1 - exp2, SObj, ExObj) = redStr(exp1, SObj, ExObj) \ redStr(exp2, SObj,
    ExObj) .

op cartesian : Set{Qid} Set{Qid} -> Set{PartialOrder} .
eq cartesian({}, SQ:Set{Qid}) = {} .
eq cartesian(SQ:Set{Qid}, {}) = {} .
eq cartesian({Q1}, {Q2}) = {(Q1, Q2)} .
eq cartesian({Q1}, {Q2, PS:PreSet{Qid}}) = union(cartesian({Q1}, {PS:PreSet{Qid}}),
    cartesian({Q1}, {Q2})) .
eq cartesian({Q1, PS:PreSet{Qid}}, SQ:Set{Qid}) = union(cartesian({PS:PreSet{Qid}}, SQ
    :Set{Qid}), cartesian({Q1}, SQ:Set{Qid})) .

op predSet : Set{PartialOrder} -> Set{Qid} .
eq predSet({}) = {} .
eq predSet({(Q1,Q2)}) = {Q1} .
eq predSet({(Q1,Q2), PS:PreSet{PartialOrder}}) = union(predSet({PS:PreSet{PartialOrder
    }}), {Q1}) .

op sucSet : Set{PartialOrder} -> Set{Qid} .
eq sucSet({}) = {} .
eq sucSet({(Q1, Q2)}) = {Q2} .
eq sucSet({(Q1,Q2), PS:PreSet{PartialOrder}}) = union(sucSet({PS:PreSet{PartialOrder
    }}), {Q2}) .

op pred : Qid Set{PartialOrder} -> Set{Qid} .
eq pred(Q:Qid, {}) = {} .
eq pred(Q:Qid, {(Q1:Qid,Q2:Qid)}) = if Q:Qid == Q2:Qid then {Q1:Qid} else {} fi .
eq pred(Q:Qid, {(Q1:Qid,Q2:Qid), PS:PreSet{PartialOrder}}) = if Q:Qid == Q2:Qid then
    union(pred(Q:Qid, {PS:PreSet{PartialOrder}}), {Q1:Qid}) else union(pred(Q:Qid, {PS
    :PreSet{PartialOrder}}), {}) fi .

op unionIfPred : Qid Set{Qid} Set{PartialOrder} -> Set{Qid} .
eq unionIfPred(Q, SQ, StSet) = if Q in predSet(StSet) then union(SQ, {Q}) else SQ fi .

op transform-in-grammar : LearnCourse -> SModule .
eq transform-in-grammar(course on Cid teaches SObj and Lid with CS LOdecls) =
        mod qid(Cid) is
                (including 'BOOL . § including 'IM-GRAMMAR .)
                sorts none .
                none --- subsort declarations
                ---op declarations:generateOps(SObj, Lid, CS)
                none
                none --- membership axioms
                none --- equations
```

```
                --- rules:
                generateRls((SObj, Lid), CS)
        endm
.

op generateRls : SetObj CourseSpecs -> RuleSet .
eq generateRls(SObj, CS exercise Lid CS') = verifyDefaultStr(SObj, Lid, CS CS') .
eq generateRls(SObj, CS exercises ExObj and Lid CS') = verifyDefaultStr(SObj, (ExObj,
    Lid), CS CS') .
eq generateRls(SObj, CS) = verifyDefaultStr(SObj, none, CS) [owise] .

op verifyDefaultStr : SetObj SetObj CourseSpecs -> RuleSet .
eq verifyDefaultStr(SObj, ExObj, none) = (rl 'S-[upTerm('default)] => 'epsilon.Word [
    none] .) § makeInitialRls(set(SObj), 'default, {}) § genStrRls(set(SObj), set(SObj)
    , 'default, {}) .
eq verifyDefaultStr(SObj, ExObj, CS) = generateRlsEX(SObj, ExObj, CS) [owise] .

op generateRlsEX : SetObj SetObj CourseSpecs -> RuleSet .
eq generateRlsEX(SObj, ExObj, CS teaching strategy Lid SStr CS') = (rl 'S-[upTerm(qid(
    Lid))] => 'epsilon.Word [none] .) § makeInitialRls(set(SObj) \ sucSet(strSet(SStr,
     SObj, ExObj)), qid(Lid), strSet(SStr, SObj, ExObj)) § genStrRls(set(SObj), set(
    SObj), qid(Lid), strSet(SStr, SObj, ExObj)) § generateRlsEX(SObj, ExObj, CS CS') .
eq generateRlsEX(SObj, ExObj, CS) = none [owise] .

op genStrRls : Set{Qid} Set{Qid} Qid Set{PartialOrder} -> RuleSet .
eq genStrRls({}, Choices, StrID, StSet) = none .
eq genStrRls({Q}, Choices, StrID, StSet) = genLOStrRls(Q, 2^(predSet(StSet)), Choices,
     StrID, StSet) .
eq genStrRls({Q, PS:PreSet{Qid}}, Choices, StrID, StSet) = genLOStrRls(Q, 2^(predSet(
    StSet)), Choices, StrID, StSet) § genStrRls({PS:PreSet{Qid}}, Choices, StrID,
    StSet) .

op genLOStrRls : Qid Set{Qid} Set{Qid} Qid Set{PartialOrder} -> RuleSet .
eq genLOStrRls(LO, {}, Choices, StrID, StSet) = none .
eq genLOStrRls(LO, {SQ}, Choices, StrID, StSet) = genLOStSetStrRls(LO, SQ, Choices,
    StrID, StSet) .
eq genLOStrRls(LO, {SQ, PS:PreSet{Qid}}, Choices, StrID, StSet) = genLOStSetStrRls(LO,
     SQ, Choices, StrID, StSet) § genLOStrRls(LO, {PS:PreSet{Qid}}, Choices, StrID,
    StSet) .

op genLOStSetStrRls : Qid Set{Qid} Set{Qid} Qid Set{PartialOrder} -> RuleSet .
eq genLOStSetStrRls(LO, SQ, {}, StrID, StSet) = none .
eq genLOStSetStrRls(LO, SQ, {Q}, StrID, StSet) = makeRls(LO, SQ, Q, StrID, StSet) .
eq genLOStSetStrRls(LO, SQ, {Q, PS:PreSet{Qid}}, StrID, StSet) = makeRls(LO, SQ, Q,
    StrID, StSet) § genLOStSetStrRls(LO, SQ, {PS:PreSet{Qid}}, StrID, StSet) .

op makeRls : Qid Set{Qid} Qid Qid Set{PartialOrder} -> RuleSet .
ceq makeRls(LO, SQ, Q, StrID, StSet) = (rl ''[_:_:_'][upTerm(LO) > upTerm(unionIfPred(
    LO, SQ, StSet)) > upTerm(StrID)] => '__[upTerm(Q) > (''[_:_:_'][upTerm(Q) > upTerm
```

```
        (unionIfPred(Q, unionIfPred(Q, unionIfPred(LO, SQ, StSet), StSet), StSet)) >
        upTerm(StrID)])] [none] . § rl ''[_:_:_'][upTerm(LO) > upTerm(unionIfPred(LO, SQ,
        StSet)) > upTerm(StrID)] => upTerm(Q) [none] .) if LO =/= Q /\ pred(LO, StSet)
        subset SQ /\ pred(Q, StSet) subset unionIfPred(LO, SQ, StSet) .
    eq makeRls(LO, SQ, Q, StrID, StSet) = none [owise] .

    op makeInitialRls : Set{Qid} Qid Set{PartialOrder} -> RuleSet .
    eq makeInitialRls({}, StrID, StSet) = none .
    eq makeInitialRls({Q}, StrID, StSet) = (rl 'S-[upTerm(StrID)] => '__[upTerm(Q) > (''[_
        :_:_'][upTerm(Q) > upTerm(unionIfPred(Q, {}, StSet)) > upTerm(StrID)])] [none] .
        § rl 'S-[upTerm(StrID)] => upTerm(Q) [none] .) .
    eq makeInitialRls({Q, PS:PreSet{Qid}}, StrID, StSet) = (rl 'S-[upTerm(StrID)] => '__[
        upTerm(Q) > (''[_:_:_'][upTerm(Q) > upTerm(unionIfPred(Q, {}, StSet)) > upTerm(
        StrID)])] [none] . § rl 'S-[upTerm(StrID)] => upTerm(Q) [none] .) § makeInitialRls
        ({PS:PreSet{Qid}}, StrID, StSet) .
endm

fmod GRAMMAR-SIG is
    inc GRAMMAR .
    sorts Production ProductionSet .
    subsort Production < ProductionSet .
    subsort Qid < V .
    op S : -> V .
    op [_,_] : Qid Set{Qid} -> V .
    op _->_ : Word Word -> Production .
    op none : -> ProductionSet .
    op __ : ProductionSet ProductionSet -> ProductionSet [assoc comm id: none format (d n
        d)] .
endfm

view NonTerminal from TRIV to GRAMMAR is
    sort Elt to V .
endv

mod CMD is
    inc GRAMMAR .
    inc GRAMMAR-SIG .
    inc GRAMMAR_TRANSFORM .
    inc MAP{NonTerminal,Qid} .
    inc CONVERSION .

    var LC : LearnCourse . var M : Module . vars Q Str : Qid . var N : Nat . vars T T' T1
        T2 : Term . var RS : RuleSet . var MP : Map{NonTerminal,Qid} . var SORT : Sort .
        var Wd : Word .

    op error : -> [Word] .
    op qEr : -> [Qid] .

    op genWord : LearnCourse Qid Nat -> Word .
```

```
op genWord : Module Qid Nat -> Word .
eq genWord(LC, Str, N) = downTerm(getTerm(metaSearch(transform-in-grammar(LC), 'S-[
    upTerm(Str)], 'W:Word, nil, '!, unbounded, N)), error) .
eq genWord(M, Str, N) = downTerm(getTerm(metaSearch(M, 'S-[upTerm(Str)], 'W:Word, nil,
     '!, unbounded, N)), error) .

op getGrammar : LearnCourse Qid -> ProductionSet .
op getGrammar : Module Qid -> ProductionSet .
eq getGrammar(LC, Str) = $grammar(getRls(transform-in-grammar(LC)), Str).
eq getGrammar(M, Str) = $grammar(getRls(M), Str).

op getLabeledGrammar : LearnCourse Qid -> ProductionSet .
op getLabeledGrammar : Module Qid -> ProductionSet .
eq getLabeledGrammar(LC, Str) = $labeledGrammar(getRls(transform-in-grammar(LC)), Str,
     empty, 0).
eq getLabeledGrammar(M, Str) = $labeledGrammar(getRls(M), Str, empty, 0).

op hasWord : LearnCourse Qid Word -> Bool .
op hasWord : Module Qid Word -> Bool .
eq hasWord(LC, Str, Wd) = metaSearch(transform-in-grammar(LC), 'S-[upTerm(Str)],
    upTerm(Wd), nil, '!, tam(Wd), 0) =/= (failure).ResultTriple? .
eq hasWord(M, Str, Wd) = metaSearch(M, 'S-[upTerm(Str)], upTerm(Wd), nil, '!, tam(Wd),
     0) =/= (failure).ResultTriple? .

op $grammar : RuleSet Qid -> ProductionSet .
eq $grammar(none, Str) = none .
eq $grammar((rl T1 => T2 [none] .) § RS, Str) = if $termInStr(Str, T1) then ($simplify(
    downTerm(T1, error)) -> $simplify(downTerm(T2, error))) $grammar(RS, Str) else
    $grammar(RS, Str) fi .

op $termInStr : Qid Term -> Bool .
eq $termInStr(Str, 'S-[T]) = T == upTerm(Str) .
eq $termInStr(Str, ''[_:_:_'][T1 > T2 > T]) = T == upTerm(Str) .

op $simplify : Word -> Word .
eq $simplify(S-(Str)) = S .
eq $simplify(Q) = Q .
eq $simplify(epsilon) = epsilon .
eq $simplify([Q : SQ:Set{Qid} : Str]) = [Q, SQ:Set{Qid}] .
eq $simplify(Q [Q : SQ:Set{Qid} : Str]) = Q $simplify([Q : SQ:Set{Qid} : Str]) .

op $labeledGrammar : RuleSet Qid Map{NonTerminal,Qid} Nat -> ProductionSet .
eq $labeledGrammar(none, Str, MP, N) = none .
eq $labeledGrammar((rl T1 => T2 [none] .) § RS, Str, MP, N) = $red1(Str, T1, T2, MP, N,
     RS) .

op $red1 : Qid Term Term Map{NonTerminal,Qid} Nat RuleSet -> ProductionSet .
ceq $red1(Str, 'S-[Q], T2, MP, N, RS) = $red2(Str, S, T2, MP, N, RS) if Q == upTerm(
    Str).
```

```
        ceq $red1(Str, T1, T2, MP, N, RS) = if $hasMapping(MP, downTerm(T1, error)) then $red2
            (Str, MP[ downTerm(T1, error)], T2, MP, N, RS) else $red2(Str, qid("v" + string(N,
             10)), T2, insert(downTerm(T1, error), qid("v" + string(N,10)),MP), N + 1, RS) fi
            if $strTerm(T1) == Str .
        eq $red1(Str, T1, T2, MP, N, RS) = $labeledGrammar(RS, Str, MP, N) [owise] .

        op $red2 : Qid Word Term Map{NonTerminal,Qid} Nat RuleSet -> ProductionSet .
        eq $red2(Str, Wd, Q, MP, N, RS) = (Wd -> downTerm(Q, qEr)) $labeledGrammar(RS, Str, MP
            , N) .
        eq $red2(Str, Wd, '__[Q > T2], MP, N, RS) = if $hasMapping(MP, downTerm(T2, error))
            then (Wd -> downTerm(Q, qEr) MP[downTerm(T2, error)]) $labeledGrammar(RS, Str, MP,
             N) else (Wd -> downTerm(Q, qEr) qid("v" + string(N,10))) $labeledGrammar(RS, Str,
             insert(downTerm(T2, error),qid("v" + string(N,10)),MP), N + 1) fi .
        eq $red2(Str, Wd, T2, MP, N, RS) = $labeledGrammar(RS, Str, MP, N) [owise] .

        op $strTerm : Term -> Qid .
        eq $strTerm(''[_:_:_'][T1 > T2 > Q]) = downTerm(Q, qEr) .
endm
```

B.3 Transformador de Learn para Teorias de Reescrita e páginas HTML

```
fmod LEARN is
        pr NAT . pr STRING . pr FLOAT . pr QID .
        sorts LearningObject Strategy .
        op lo : String -> LearningObject [ctor] .
        op st : String -> Strategy [ctor] .
        ops title text image : LearningObject -> String .
        var L : LearningObject .
        eq text(L) = "" [owise] .
        eq image(L) = "" [owise] .
endfm

fmod LO-LIST is
        pr LEARN .
        sort LOList .
        subsort LearningObject < LOList .
        op mtlist : -> LOList .
        op _;_ : LOList LOList -> LOList [assoc id: mtlist] .
endfm

mod LO-SET is
        pr LEARN .
        sort LOSet .
        subsort LearningObject < LOSet .
        op mtset : -> LOSet .
        op __ : LOSet LOSet -> LOSet [assoc comm id: mtset] .
```

```
        op choose : LOSet ~> LearningObject .
        op _-_ : LOSet LOSet -> LOSet .
        op _diff_ : LOSet LOSet ~> LearningObject .
        op _subset_ : LOSet LOSet -> Bool .
        var A : LearningObject . vars AS AS' : LOSet .
        eq (AS AS') - AS = AS' .
        eq AS - AS' = AS [owise] .
        eq (A AS) diff AS = A .
        eq AS subset (AS AS') = true .
        eq AS subset AS' = false [owise] .
        rl choose(A AS) => A .
endm

mod STATE is
        pr LO-LIST . pr LO-SET .
        sort State .
        op state : Strategy LOSet LOList -> State .
endm

mod STRATEGY is
        pr STATE .
        op init : Strategy -> State .
        ops all exercises : -> LOSet .
        op pred : Strategy LearningObject -> LOSet .
        op _contained_ : LOSet LOList -> Bool .
        vars A B : LearningObject . vars AL L1 L2 : LOList . var AS : LOSet . var S : Strategy
            .
        eq init(S) = state(S, all, mtlist) .
        eq mtset contained AL = true .
        ceq (A AS) contained (L1 ; A ; L2) = true if AS contained (L1 ; L2) .
        eq AS contained AL = false [owise] .
        eq pred(S, LO:LearningObject) = mtset [owise] .
        crl state(S, AS, mtlist) => state(S, (AS - A), A) if choose(AS) => A /\ pred(S, A) =
            mtset .
        crl state(S, AS, AL) => state(S, (AS - A) B, AL ; A) if choose(AS) => A /\ B := all
            diff AS /\ pred(S, A) contained AL /\ AL =/= mtlist .
endm

mod TRANSFORM-LEARN is
        inc LEARN-SIG .
        inc STRATEGY .
        inc META-LEVEL .

        var S : String . var SObj : SetObj . var Lid : LearnID . var Cid : CourseID . var Str
            : StrDecl .
        var LOdecls : LearnObjDecls . var CS : CourseSpecs . var Q : Qid . var SStr : SetStr .
        var OAt : ObjAttrib . var SAt : SetLOAttrib . vars Ext1 Ext2 : StrExp .

        op str : LearnID -> String .
```

```
eq str(< Q >) = string(Q) .
op q : CourseID -> Qid .
eq q(S) = qid(S) .

op transformLearnCourse : LearnCourse -> SModule .
eq transformLearnCourse(course on Cid teaches SObj and Lid with CS LOdecls) =
        mod q(Cid) is
                (including 'BOOL . including 'STRATEGY .)
                sorts none .
                none
                none
                none
                (eq 'all.LOSet = transformOverSetObj(SObj, Lid) [none] .)
                 transformCourseSpecs(CS) transformLearnObjDecls(LOdecls)
                none
        endm
.

op transformOverSetObj : SetObj LearnID -> Term .
eq transformOverSetObj(Lid, none) = transformLearnID(Lid) .
eq transformOverSetObj(SObj, Lid) = '__[transformSetObj(SObj), transformLearnID(Lid)]
    [owise] .

op transformSetObj : SetObj -> Term .
eq transformSetObj(Lid) = transformLearnID(Lid) .
eq transformSetObj(Lid, SObj) = '__[transformLearnID(Lid), transformSetObj(SObj)] .

op transformLearnID : LearnID -> Term .
eq transformLearnID(Lid) = 'lo[upTerm(str(Lid))] .

op transformCourseSpecs : CourseSpecs -> EquationSet .
eq transformCourseSpecs(none) = none .
eq transformCourseSpecs(exercise Lid CS) = (eq 'exercises.LOSet = transformLearnID(Lid
    ) [none] .) transformCourseSpecs(CS) .
eq transformCourseSpecs(exercises SObj and Lid CS) = (eq 'exercises.LOSet =
    transformOverSetObj(SObj, Lid) [none] .) transformCourseSpecs(CS) .
eq transformCourseSpecs(teaching strategy Lid SStr CS) = transformSetStr(Lid, SStr)
    transformCourseSpecs(CS) .

op transformSetStr : LearnID SetStr -> EquationSet .
eq transformSetStr(Lid, Str) = transformStrDecl(Lid, Str) .
eq transformSetStr(Lid, Str, SStr) = transformStrDecl(Lid, Str) transformSetStr(Lid,
    SStr) .

op transformStrDecl : LearnID StrDecl -> EquationSet .
eq transformStrDecl(Lid, Ext1 after Ext2) = (ceq 'pred['st[upTerm(str(Lid))], 'LO:
    LearningObject] = transformStrExp(Ext2) if '_subset_['LO:LearningObject,
    transformStrExp(Ext1)] = 'true.Bool [none] .) .
eq transformStrDecl(Lid, Ext1 before Ext2) = (ceq 'pred['st[upTerm(str(Lid))], 'LO:
```

```
        LearningObject] = transformStrExp(Ext1) if '_subset_['LO:LearningObject,
        transformStrExp(Ext2)] = 'true.Bool [none] .) .

    op transformStrExp : StrExp -> Term .
    eq transformStrExp(all) = 'all.LOSet .
    eq transformStrExp(exercises) = 'exercises.LOSet .
    eq transformStrExp(Lid) = transformLearnID(Lid) .
    eq transformStrExp(Ext1 - Ext2) = '_-_[transformStrExp(Ext1), transformStrExp(Ext2)] .

    op transformLearnObjDecls : LearnObjDecls -> EquationSet .
    eq transformLearnObjDecls(LOdecl:LearnObjDecl) = transformLearnObjDecl(LOdecl:
        LearnObjDecl) .
    eq transformLearnObjDecls(LOdecl:LearnObjDecl LOdecls) = transformLearnObjDecl(LOdecl:
        LearnObjDecl) transformLearnObjDecls(LOdecls) .

    op transformLearnObjDecl : LearnObjDecl -> EquationSet .
    eq transformLearnObjDecl(learning object Lid has title S) = (eq 'title['lo[upTerm(str(
        Lid))]] = upTerm(S) [none] .) .
    eq transformLearnObjDecl(learning object Lid has title S SAt) = (eq 'title['lo[upTerm(
        str(Lid))]] = upTerm(S) [none] .) transformSetLOAttrib(Lid, SAt) .

    op transformSetLOAttrib : LearnID SetLOAttrib -> EquationSet .
    eq transformSetLOAttrib(Lid, OAt) = transformObjAttrib(Lid, OAt) .
    eq transformSetLOAttrib(Lid, OAt SAt) = transformObjAttrib(Lid, OAt)
        transformSetLOAttrib(Lid, SAt) .

    op transformObjAttrib : LearnID ObjAttrib -> Equation .
    eq transformObjAttrib(Lid, text S) = (eq 'text['lo[upTerm(str(Lid))]] = upTerm(S) [
        none] .) .
    eq transformObjAttrib(Lid, image S) = (eq 'image['lo[upTerm(str(Lid))]] = upTerm(S) [
        none] .) .
endm

mod toHTML is
    pr TRANSFORM-LEARN .

    var module : SModule . vars T T' TS : Term . var AS : AttrSet .
    var EC : EqCondition . vars S S1 S2 : String . vars L LOS PredSet : LOSet .
    vars SL SL2 : StringList . var LO : LearningObject . var EqtS : EquationSet .
    op error : -> [LOSet] .
    op error : -> [String] .

    sorts StringList SSet .
    subsort String < StringList .
    op mtlist : -> StringList .
    op _,_ : StringList StringList -> StringList [assoc comm id: mtlist] .
    op mtset : -> SSet .
    op <_> : StringList -> SSet .
    op _+_ : SSet SSet -> SSet [assoc comm id: mtset] .
```

```
eq < mtlist > = mtset .
eq < S , S, SL > = < S, SL > .
eq < SL > + < SL2 > = < SL , SL2 > .

op setToString : SSet -> String .
eq setToString(mtset) = "<script type=\"text/javascript\">init(\"default\")</script>"
    .
eq setToString(< S >) =  "<script type=\"text/javascript\">init(\"" + S:String + "\")
    </script>" .
ceq setToString(< S , SL >) = "Estrategia:<br>\n" + makeStrString(< S , SL >) if SL:
    StringList =/= mtlist .
op makeStrString : SSet -> String .
eq makeStrString(mtset) = "" .
eq makeStrString(< S , SL >) = "<a href=\"#\" onclick='init(\"" + S:String + "\");
    return false;'>" + S:String + "</a><br>\n" + makeStrString(< SL >) .

op convertHTML : SModule -> String .
eq convertHTML(module) = "<html>\n<head>\n<title>" + string(getName(module)) + "</
    title>\n<script type=\"text/javascript\">\n" + "var list = []; var strategy;\
    nfunction contained(set){\nfor (var i = 0; i < set.length; i++)\nif(list.
    lastIndexOf(set[i]) == -1) return false;\nreturn true;\n}\n" + "function init(str)
    {\nstrategy = str;\nvar aux = window.localStorage.getItem('learn_list_'+strategy)
    ;\nif(aux)\nlist = JSON.parse(aux);\ndocument.body.innerHTML = '<center><b><big><
    big>" + string(getName(module)) + "</big></big></b></center><br>' + links('');\n}\
    n" + "function pred(lo){\n" + convertPredEqs(module, getEqs(module)) + "return
    [];\n}\n" + "function links(last){\nvar linkList = \"\";\n" + createLinks(module,
    downTerm(getTerm(metaReduce(module, 'all.LOSet)), error)) + "return linkList;\n}\n
    " + createLOFunctions(module, downTerm(getTerm(metaReduce(module, 'all.LOSet)),
    error)) + "</script>\n</head>\n<body>\n" + insertStr(getEqs(module)) + "\n</body>\
    n</html>\n" .

op convertPredEqs : Module EquationSet -> String .
eq convertPredEqs(module, none) = "" .
eq convertPredEqs(module, eq T = T' [AS] . EqtS) = convertPredEqs(module, EqtS) .
eq convertPredEqs(module, ceq 'pred['st[TS], 'LO:LearningObject] = T if '_subset_['LO:
    LearningObject, T'] = 'true.Bool [AS] . EqtS) = createIF(downTerm(TS, error),
    downTerm(getTerm(metaReduce(module, T')), error), downTerm(getTerm(metaReduce(
    module, T)), error)) + convertPredEqs(module, EqtS) .

op createIF : String LOSet LOSet -> String .
eq createIF(S, mtset, PredSet) = "" .
eq createIF(S, lo(S1) LOS, PredSet) = "if(strategy == '" + S + "' && lo == '" + S1 +
    "') return [" + convertPredSet(PredSet) + "];\n" + createIF(S, LOS, PredSet) .

op convertPredSet : LOSet -> String .
eq convertPredSet(mtset) = "" .
eq convertPredSet(lo(S)) = "'" + S + "'" .
eq convertPredSet(lo(S) LOS) = "'" + S + "'," + convertPredSet(LOS) .
```

```
        op createLinks : Module LOSet -> String .
        eq createLinks(module, mtset) = "" .
        eq createLinks(module, lo(S) LOS) = "if(last != '" + S + "' && contained(pred('" + S +
             "'))) linkList += '<br><a href=\"#\" onclick=\"" + S + "();return false\">" +
            downTerm(getTerm(metaReduce(module, 'title['lo[upTerm(S)]])), error) + "</a>';\n"
            + createLinks(module, LOS) .

        op createLOFunctions : Module LOSet -> String .
        eq createLOFunctions(module, mtset) = "" .
        eq createLOFunctions(module, lo(S) LOS) = "function " + S + "(){\nlist.push('" + S +
            "');\nwindow.localStorage.setItem('learn_list_'+strategy, JSON.stringify(list));\
            nvar prt = '<div style=\"text-align: center;\"><b><big><big>" + downTerm(getTerm(
            metaReduce(module, 'title['lo[upTerm(S)]])), error) + "</big></big></b></div><br>"
             + insertImg(module, lo(S)) + insertTxt(module, lo(S)) + "';\nprt += links('" + S
            + "');\ndocument.body.innerHTML = prt;\n}\n" + createLOFunctions(module, LOS)
        .

        op insertImg : Module LearningObject -> String .
        ceq insertImg(module, lo(S)) = "<br><img src=\"" + downTerm(getTerm(metaReduce(module,
             'image['lo[upTerm(S)]])), error) + "\"><br>" if downTerm(getTerm(metaReduce(
            module, 'image['lo[upTerm(S)]])), error) =/= "" .
        eq insertImg(module, LO) = "" [owise] .

        op insertTxt : Module LearningObject -> String .
        ceq insertTxt(module, lo(S)) = downTerm(getTerm(metaReduce(module, 'text['lo[upTerm(S)
            ]])), error) + "<br>" if downTerm(getTerm(metaReduce(module, 'text['lo[upTerm(S)
            ]])), error) =/= "" .
        eq insertTxt(module, LO) = "" [owise] .

        op insertStr : EquationSet -> String .
        eq insertStr(EqtS) = setToString(collectStr(EqtS)) .

        op collectStr : EquationSet -> SSet .
        eq collectStr(none) = mtset .
        eq collectStr(eq T = T' [AS] . EqtS) = collectStr(EqtS) .
        eq collectStr(ceq 'pred['st[TS], 'LO:LearningObject] = T if '_subset_['LO:
            LearningObject, T'] = 'true.Bool [AS] . EqtS) = < downTerm(TS, error) > +
            collectStr(EqtS) .
endm

mod CMD is
        inc TRANSFORM-LEARN .
        inc toHTML .

        op st : -> [State] .
        op lObj : -> [LearningObject] .
        op loSet : -> [LOSet] .
        op loList : -> [LOList] .
```

```
        var LC : LearnCourse . var T : Term . var Q : Qid .
        var B : Bound . var N : Nat . var S : State .

        op transform : LearnCourse -> Module .
        eq transform(LC) = transformLearnCourse(LC) .
        op reduce : LearnCourse Term -> Term .
        eq reduce(LC, T) = getTerm(metaReduce(transform(LC), T)) .
        op search : LearnCourse Qid Bound Nat -> State .
        eq search(LC, Q, B, N) = downTerm(getTerm(metaSearch(transform(LC), 'init['st[upTerm(
            string(Q))]], 'S:State, nil, '*, B, N)), st) .
        op search : LearnCourse Qid Nat -> State .
        eq search(LC, Q, N) = search(LC, Q, unbounded, N) .
        op dsearch : LearnCourse Qid Nat -> State .
        eq dsearch(LC, Q, N) = search(LC, Q, N) .
        op dsearch : LearnCourse Nat -> State .
        eq dsearch(LC, N) = search(LC, 'default, N) .
        op path : LearnCourse Qid Bound Nat -> Trace .
        eq path(LC, Q, B, N) = metaSearchPath(transform(LC), 'init['st[upTerm(string(Q))]], 'S
            :State, nil, '*, B, N) .
        op path : LearnCourse Qid Nat -> Trace .
        eq path(LC, Q, N) = path(LC, Q, unbounded, N) .

        op html : LearnCourse -> String .
        eq html(LC) = convertHTML(transform(LC)) .
endm
```