

UNIVERSIDADE FEDERAL FLUMINENSE

SEMANA DE ENGENHARIA 2009

Introdução às Linguagens NCL e Lua: Desenvolvendo Aplicações Interativas para TV Digital

Autores:

Rafael Carvalho¹

Joel André Ferreira dos Santos²

Jean Ribeiro Damasceno³

Julia Varanda da Silva⁴

Débora Christina Muchaluat Saade⁵

E-mails: rafael.carvalho@peta5.com.br, (joel, damascon, julia, debora)@midiacom.uff.br

19 de outubro de 2009

¹Peta5

²Laboratório MídiaCom, Peta5, Departamento de Engenharia de Telecomunicações

³Laboratório MídiaCom, Departamento de Engenharia de Telecomunicações

⁴Peta5, Laboratório MídiaCom, Instituto de Computação

⁵Laboratório MídiaCom, Instituto de Computação

Sumário

1	Introdução	3
2	Middleware	6
2.1	Linguagem Declarativa X Linguagem Procedural	7
2.1.1	Linguagens Declarativas	7
2.1.2	Linguagens Procedurais	7
2.1.3	Aplicações Híbridas	7
3	Linguagem NCL	10
3.1	Introdução	10
3.1.1	Nested Context Model	10
3.1.2	Onde	10
3.1.3	Como	11
3.1.4	O que	11
3.1.5	Quando	11
3.2	Extensible Markup Language	12
3.3	Estrutura de um documento NCL	13
3.3.1	Cabeçalho	14
3.3.2	Corpo	14
3.4	Definindo a apresentação	14
3.4.1	Regiões	14
3.4.2	Descritores	16
3.4.3	Prática	17
3.5	Inserindo os elementos	17
3.5.1	Mídias	18
3.5.2	Âncoras	18
3.5.3	Propriedades	19
3.5.4	Prática	20
3.6	Organizando o documento	20
3.6.1	Contexto	20
3.6.2	Portas	21
3.6.3	Prática	21
3.7	Sincronizando os elementos	21
3.7.1	Conectores	22
3.7.2	Elos	24
3.7.3	Prática	25
3.8	Definindo alternativas	28
3.8.1	Regras	28

3.8.2	Switch	28
4	Linguagem Lua	31
4.1	Introdução	31
4.1.1	Pequeno histórico	31
4.1.2	Convenções Léxicas	31
4.1.3	O interpretador de linha de comando	32
4.1.4	Prática	32
4.2	Tipos e variáveis	33
4.2.1	Tipos	34
4.2.2	Variáveis	35
4.2.3	Prática	35
4.3	Operadores	36
4.3.1	Aritiméticos	36
4.3.2	Relacionais	36
4.3.3	Lógicos	37
4.3.4	Concatenação	38
4.3.5	Prática	38
4.4	Estruturas de controle	39
4.4.1	if then else	39
4.4.2	while	40
4.4.3	repeat	40
4.4.4	for numérico	40
4.4.5	for genérico	41
4.5	Saindo de um bloco	42
4.6	Funções	42
4.6.1	Declaração e execução	43
4.6.2	Número variável de argumentos	43
4.6.3	Retorno	44
4.6.4	Prática	44
4.7	Tabelas	45
4.7.1	Criação de tabelas	45
4.7.2	Tamanho de tabelas	46
4.7.3	Prática	46
5	Integração NCL-Lua	49
5.1	Introdução	49
5.2	Scripts NCLua	49
5.3	Módulos NCLua	49
5.3.1	Módulo event	50
5.3.2	Módulo canvas	53
5.3.3	Módulo settings	60
5.3.4	Módulo persistent	60
A	Exemplos da Apostila	62
B	Ferramentas	71
B.1	Composer	71
B.2	NCL Eclipse	74

Capítulo 1

Introdução

A TV digital interativa traz inúmeras facilidades e oportunidades para os programas de televisão. Com o uso deste sistema, os telespectadores podem ter acesso a jogos e programas interativos cujo conteúdo pode ser moldado de acordo com a sua escolha.

O recente sistema brasileiro de TV digital [1] foi definido de forma a possibilitar a criação de programas com as características citadas.

O sistema brasileiro de TV digital possui uma arquitetura em camadas, onde cada camada oferece serviços para a camada imediatamente superior e usa os serviços da camada imediatamente inferior, como ilustrado na Figura 2.1.

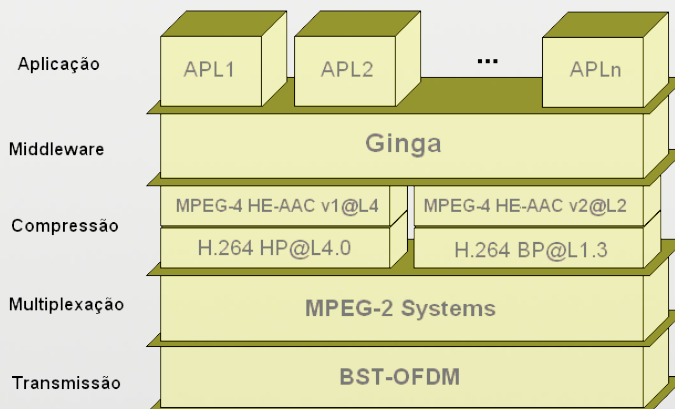


Figura 1.1: Arquitetura do Sistema Brasileiro de TV Digital.

As camadas são caracterizadas da seguinte forma:

- Serviços, aplicações e conteúdo: camada responsável pela captura e formatação dos sinais de áudio e vídeo, bem como a implementação de serviços interativos na forma de software para serem executadas em uma ou mais entidades de hardware [2].
- Middleware: camada de software que realiza a integração de todas as subcamadas do sistema. O middleware permite que os aplicativos gerados pelas emissoras sejam compatíveis com todas as plataformas de recepção desenvolvidas para o padrão de TV digital [3].
- Compressão: camada responsável pela remoção de redundâncias nos sinais de áudio e vídeo, reduzindo assim a taxa de bits necessária para transmitir essas informações [2]. No receptor, essa camada decodifica os fluxos de áudio e vídeo recebidos. O sistema brasileiro

adotou o padrão MPEG-4 HE AAC para compressão de áudio e o padrão H.264 para compressão de vídeo.

- Multiplexação: camada responsável por gerar um único fluxo de dados contendo o vídeo, áudio e aplicações dos diversos programas a serem transmitidos, baseada no padrão MPEG-2 Systems [2].
- Transmissão/Recepção: também denominada de camada física, é a camada responsável por levar as informações digitais da emissora até a casa do telespectador [4]. O sistema brasileiro adotou a mesma tecnologia utilizada no sistema japonês de TV digital, baseada em BST-OFDM.

Os programas, ou aplicações, para o ambiente de TV digital podem ser criados com o uso de dois paradigmas de programação distintos: declarativo e procedural. O middleware Ginga [5], adotado como padrão no sistema brasileiro, é subdividido em Ginga-NCL [3, 6], que dá suporte às aplicações declarativas e Ginga-J [8], que oferece suporte às aplicações que utilizam o paradigma procedural.

Esta apostila apresenta o middleware Ginga, focando em sua parte declarativa Ginga-NCL, que permite o uso das linguagens NCL (Nested Context Language) [6] e Lua [7] para autoria de aplicações para o ambiente de TV digital.

O Capítulo 2 apresenta uma introdução ao middleware Ginga e discute diferenças entre os paradigmas de programação declarativo e procedural.

O Capítulo 3 introduz a linguagem NCL apresentando os principais elementos da linguagem através de exemplos práticos.

O Capítulo 4 aborda a linguagem Lua e também utiliza exemplos para discutir suas principais características.

O Capítulo 5 discute a integração entre as linguagens NCL e Lua em uma mesma aplicação para TV digital.

Os Apêndices A e B apresentam os códigos fonte dos exemplos completos discutidos ao longo do texto e também abordam as principais ferramentas de autoria de aplicações para o Ginga-NCL.

Referências

- [1] Norma ABNT 15606-2:2007. Televisão digital terrestre - Codificação de dados e especificações de transmissão para radiodifusão digital - Parte 2: Ginga-NCL para receptores fixos e móveis - Linguagem de aplicação XML para codificação de aplicações.
- [2] Marcelo Sampaio de Alencar. Televisão Digital. Editora Érica. Edição 1.
- [3] Soares, L.F.G; Rodrigues, R.F; Moreno, M.F. Ginga-NCL: The Declarative Environment of the Brazilian Digital TV System. Journal of the Brazilian Computer Society. Março, 2007.
- [4] Mendes, L. L. SBTVD - Uma visão sobre a TV Digital no Brasil. Outubro, 2007. TC Amazônia, Ano V, Número 12.
- [5] Ginga, Brazilian's Digital TV Middleware. (<http://www.ginga.org.br>). Acessado em Outubro de 2009.
- [6] ITU International Telecommunication Union. Nested Context Language (NCL) and Ginga-NCL for IPTV services. ITU-T Recommendation H.761. 2009. (<http://www.itu.int/rec/T-REC-H.761-200904-P>).
- [7] R. Ierusalimschy, L. H. de Figueiredo, W. Celes. The evolution of Lua. Proceedings of ACM HOPL III (2007) 2-1-2-26.
- [8] G. Souza Filho, L. Leite, C. Batista. Ginga-J: The Procedural Middleware for the Brazilian Digital TV System. Journal of the Brazilian Computer Society. Março, 2007.

Capítulo 2

Middleware

Ginga é o nome do Middleware aberto do Sistema Brasileiro de TV Digital [1]. A arquitetura e as facilidades Ginga foram projetadas para serem aplicadas a sistemas de radiodifusão e receptores de radiodifusão terrestre e a sistemas IPTV [2]. Adicionalmente, a mesma arquitetura e facilidades podem ser aplicadas a sistemas que utilizam outros mecanismos de transporte de dados (como sistema de televisão via satélite ou a cabo).

A finalidade da camada de middleware é oferecer um serviço padronizado para as aplicações, escondendo as peculiaridades e heterogeneidades das camadas inferiores, tais como, técnicas de compressão de mídias, de transporte e de modulação. Um middleware padronizado facilita a portabilidade das aplicações, permitindo que sejam executadas em qualquer receptor digital (ou set-top box) que suporte o middleware adotado. Essa portabilidade é primordial em sistemas de TV digital, pois é impossível considerar como premissa que todos os receptores digitais sejam exatamente iguais.

A arquitetura da implementação de referência do middleware Ginga, ilustrada na Figuras 2.1, pode ser dividida em três grandes módulos: Ginga-CC (Common Core) que concentra serviços necessários tanto para a máquina de apresentação quanto para a máquina de execução; o ambiente de apresentação Ginga-NCL (declarativo), que suporta o desenvolvimento de aplicações declarativas; e o ambiente de execução Ginga-J (procedural), que suporta o desenvolvimento de aplicações procedurais. Dependendo das funcionalidades requeridas no projeto de cada aplicação, um ambiente de desenvolvimento de aplicações será mais adequado que o outro.



Figura 2.1: Arquitetura do middleware Ginga.

Uma aplicação de TV Digital pode ser declarativa, procedural ou híbrida. As seções a seguir apresentam esses conceitos.

2.1 Linguagem Declarativa X Linguagem Procedural

2.1.1 Linguagens Declarativas

As linguagens declarativas são mais intuitivas (de mais alto nível) e, por isso, mais fáceis de usar. Elas normalmente não exigem um perito em programação. O programador fornece apenas o conjunto de tarefas a serem realizadas, não estando preocupado com os detalhes de como o executor da linguagem (interpretador, compilador ou a própria máquina real ou virtual de execução) realmente implementará essas tarefas. Em outras palavras, a linguagem enfatiza a declaração descritiva de um problema ao invés de sua decomposição em implementações algorítmicas, não necessitando, em geral, de tantas linhas de código para definir certa tarefa. Contudo, as linguagens declarativas têm de ser definidas com um foco específico. Quando o foco da aplicação não casa com o da linguagem, o uso de uma linguagem procedural não é apenas vantajoso, mas se faz necessário.

No padrão Brasileiro, o ambiente declarativo Ginga-NCL adotou NCL (*Nested Context Language*) [3], uma aplicação XML, para a autoria de conteúdo multimídia interativo. NCL é baseada no NCM (*Nested Context Model*) [4], modelo conceitual para especificação de documentos hipermídia com sincronização temporal e espacial entre seus objetos de mídia. NCL permite ao autor descrever o comportamento espacial e temporal de uma apresentação multimídia, associar *hyperlinks* (interação do usuário) a objetos de mídia, definir alternativas para apresentação (adaptação) e descrever o leiaute da apresentação em múltiplos dispositivos.

2.1.2 Linguagens Procedurais

O uso de linguagens procedurais (imperativas) usualmente requer um perito em programação. O programador deve informar ao computador cada passo a ser executado. Pode-se afirmar que, em linguagens procedurais, o programador possui um maior controle do código, sendo capaz de estabelecer todo o fluxo de controle e execução de seu programa. Entretanto, para isso, ele deve ser bem qualificado e conhecer bem os recursos de implementação.

Linguagens procedurais para sistemas de TV digital são frequentemente usadas em tarefas, como processamento matemático, manipulação sobre textos, uso do canal de interatividade, controle fino do teclado, animações e colisões para objetos gráficos e, de maneira geral, tarefas que necessitem da especificação de algoritmos e estruturas de dados. Por outro lado, linguagens procedurais, apesar de genéricas, introduzem uma maior complexidade de programação e dependem de uma base lógica que autores de conteúdo audio-visual nem sempre possuem.

O ambiente procedural Ginga-J [5] adota a linguagem Java [6], como uma ferramenta para a criação de conteúdo interativo, tal como os sistemas de TV digital Americano, Europeu e Japonês o fizeram.

2.1.3 Aplicações Híbridas

Uma aplicação de TV digital pode ser declarativa, procedural ou híbrida. Uma aplicação híbrida é aquela cujo conjunto de entidades possui tanto conteúdo do tipo declarativo quanto procedural. Em particular, as aplicações declarativas frequentemente fazem uso de *scripts*, cujo conteúdo é de natureza procedural. Além disso, uma aplicação declarativa pode fazer referência a um código Java TV Xlet embutido. Da mesma forma, uma aplicação procedural pode fazer referência a

uma aplicação declarativa, contendo, por exemplo, conteúdo gráfico, ou pode construir e iniciar a apresentação de aplicações com conteúdo declarativo. Portanto, ambos os tipos de aplicação no middleware Ginga podem utilizar as facilidades dos ambientes de aplicação declarativo e procedural.

O ambiente Ginga-NCL também permite o uso de Lua [7], uma linguagem de *script*, dentro de aplicações NCL.

O poder de uma linguagem declarativa é elevado quando integrada com uma linguagem procedural, passando a ter acesso a recursos computacionais genéricos. Essa integração deve seguir critérios que não afetem os princípios da linguagem declarativa, mantendo uma separação bem definida entre os dois ambientes. O autor da aplicação usa a forma declarativa sempre que possível e lança mão da forma procedural somente quando necessário.

A criação da nova classe de objetos de mídia Lua, os quais são chamados de NCLua, é a principal via de integração de NCL a um ambiente procedural, conforme definido em seu perfil para TV Digital. Por meio de elementos de mídia, *scripts* NCLua podem ser inseridos em documentos NCL, trazendo poder computacional adicional às aplicações declarativas.

Referências

- [1] Norma ABNT 15606-2:2007. Televisão digital terrestre - Codificação de dados e especificações de transmissão para radiodifusão digital - Parte 2: Ginga-NCL para receptores fixos e móveis - Linguagem de aplicação XML para codificação de aplicações.
- [2] ITU International Telecommunication Union. Nested Context Language (NCL) and Ginga-NCL for IPTV services. ITU-T Recommendation H.761. 2009. (<http://www.itu.int/rec/T-REC-H.761-200904-P>).
- [3] H. Silva, D. Saade, R. Rodrigues, and L. Soares. Ncl 2.0: Integrating new concepts to xml modular languages. ACM DocEng 2004, 2004. Wisconsin.
- [4] L. Soares and R. Rodrigues. "Nested context model 3.0 part 1 – NCM Core". Monografias em Ciência da Computação do Departamento de Informática. PUC-Rio, No. 18/05. Rio de Janeiro, Maio 2005. ISSN 0103-9741.
- [5] G. Souza Filho, L. Leite, and C. Batista. Ginga-J: The Procedural Middleware for the Brazilian Digital TV System. Journal of the Brazilian Computer Society, 12(4), March 2007.
- [6] H. M. Deitel e P.J. Deitel. (2006) "Java Como Programar". Pearson Prentice Hall, 6ª edição.
- [7] F. SantAnna, R. Cerqueira, and L. Soares. NCLua - Objetos Imperativos Lua na Linguagem Declarativa NCL. XIV Brazilian Symposium on Multimedia and the Web (WebMedia 2008), pages 83–90, 2008. Porto Alegre, RS, Brazil.

Capítulo 3

Linguagem NCL

3.1 Introdução

Este capítulo descreve a linguagem NCL (Nested Context Language) [1] em sua versão 3.0. NCL é uma linguagem baseada em XML [2], destinada à autoria de documentos hipermídia. Por ser uma linguagem de colagem, NCL é utilizada para reunir objetos de mídia em uma apresentação multimídia. Estes objetos de mídia podem ser de qualquer tipo, áudio, vídeo, um documento HTML ou mesmo um objeto procedural. Para entender a linguagem NCL, bem como seu uso, se faz necessário um estudo sobre o modelo que a gerou. As seções a seguir apresentam o modelo NCM (Nested Context Model), no qual NCL é baseada.

3.1.1 Nested Context Model

Um documento hipermídia, de uma forma genérica, é composto por nós e elos. Os nós representam abstrações das mídias utilizadas no documento além de trazerem informações adicionais, como por exemplo informações sobre a sua apresentação. Os elos fazem a sincronização espacial ou temporal entre os nós que compõem o documento.

O modelo NCM (Nested Context Model) [3], Modelo de Contextos Aninhados, estende os conceitos acima aumentando o poder e flexibilidade de um documento hipermídia. NCM estende a definição de nós em dois tipos, nós de conteúdo e nós de composição. Um nó de conteúdo traz informações sobre uma mídia utilizada pelo documento, enquanto um nó de composição possui um conjunto de nós de conteúdo e/ou outros nós de composição e conjunto de elos, sendo utilizado para dar estrutura e organização a um documento hipermídia.

Na construção de um documento hipermídia, algumas informações básicas são necessárias, indicando o que deve ser apresentado, como, quando e onde deve ser apresentado. As próximas seções discutem as entidades do modelo NCM utilizadas para especificar essas informações.

3.1.2 Onde

Para que um nó em um documento hipermídia seja apresentado, é necessária a definição de uma área para a exibição do mesmo. O modelo NCM define elementos específicos para este fim, denominados **regiões**. Estes elementos indicam a posição e o tamanho da área onde nós poderão ser apresentados, sem se preocupar com a definição dos nós que efetivamente serão apresentados nesta área. A Figura 3.1, demonstra uma região.

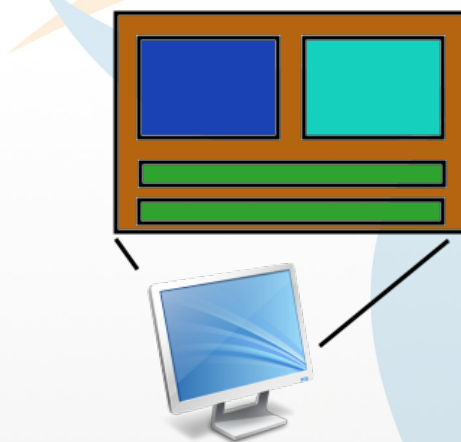


Figura 3.1: Representação de uma região.

3.1.3 Como

A definição de uma região precisa ser complementada com outras informações que indicam como cada nó será apresentado. Esta descrição completa das características de apresentação de cada nó é feita através de elementos denominados **descritores**. Um descritor pode descrever parâmetros sobre a apresentação dos nós, incluindo a região onde será apresentado, seu volume, sua transparência, a duração de sua apresentação, entre outros. A Figura 3.2, demonstra um descritor.



Figura 3.2: Representação de um descritor.

3.1.4 O que

O conteúdo de um documento hipermídia é representado através de elementos denominados **mídia**. Uma mídia representa cada nó de um documento, informando o descritor ao qual está relacionado. De acordo com o modelo NCM, uma mídia deve estar necessariamente dentro de um nó de composição, denominado **contexto**, que é utilizado para representar um documento completo ou parte dele. A Figura 3.3, demonstra o conceito de mídias e contextos.

3.1.5 Quando

Uma vez tendo escolhido os nós que farão parte do documento hipermídia, torna-se necessário definir qual será o primeiro nó a ser apresentado e a ordem de apresentação dos demais nós. Essa definição é feita com o uso de elementos chamados **portas** e elos (**links**). As portas definem os nós a serem apresentados quando um nó de contexto é iniciado e os links definem os relacionamentos de sincronização entre os nós e a interatividade do programa. Um link, entretanto, não define todo o comportamento de um relacionamento por si só, para isso é necessário o uso de **conectores**. A Figura 3.4, demonstra o uso de portas e elos.

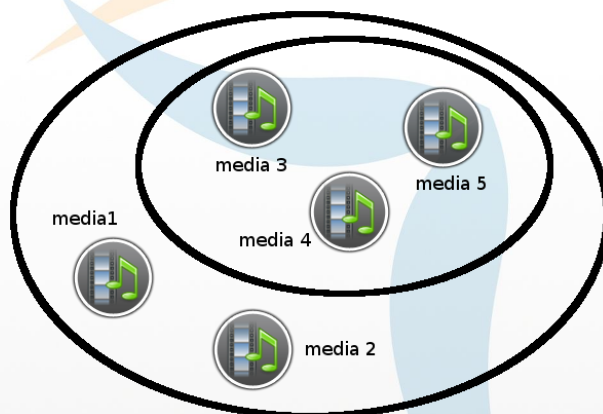


Figura 3.3: Representação de mídias e contextos.

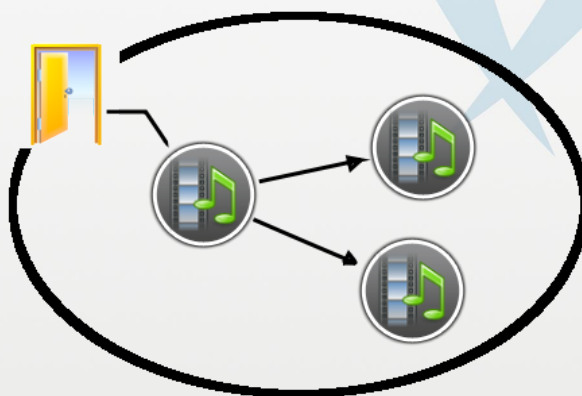


Figura 3.4: Representação de portas e elos.

3.2 Extensible Markup Language

Criada pelo World Wide Web Consortium (W3C) [4] em meados da década de 1990, a linguagem XML (Extensible Markup Language) [2] é um formato textual simples e bastante flexível criado para estruturar, armazenar e representar informação. Como XML não foi projetada para uma aplicação específica, ela pode ser usada como base (metalinguagem) para a criação de linguagens de marcação.

Em XML os documentos são organizados de forma hierárquica em uma árvore onde cada elemento possui um elemento pai e elementos filhos. Essa organização se assemelha a da árvore genealógica de uma família onde cada indivíduo seria um elemento XML. Assim como no exemplo da árvore genealógica, onde cada indivíduo possui um nome e atributos (cor de cabelo, altura, etc) que o definem, um elemento em XML também possui uma identificação e atributos. Um exemplo pode ser visto na Figura 3.7.

No padrão textual XML a definição de um elemento é iniciada pelo símbolo "<" e terminado pelos símbolos ">". Entre esses dois símbolos são definidos o nome do elemento e os atributos do mesmo. Os atributos de um elemento XML são definidos por uma dupla (nome, valor). A Figura 3.5 demonstra a definição de um elemento usado para identificar um livro.

No exemplo acima o elemento possui o nome *livro* e os atributos *autor*, *título* e *subtítulo*. Pode-se notar que o valor de cada atributo é indicado após o símbolo "=" e entre aspas. Pode-se notar também que o nome do elemento, assim como os seus atributos, não possuem letras


```
<livro autor="J. R. R. Tolkien" titulo="Senhor do Anéis"
      subtítulo="O Retorno do Rei"/>
```

Figura 3.5: Definição de um elemento.

maiúsculas nem acentos, essa é uma boa prática para evitar o surgimento de erros na utilização do documento criado, já que XML é case sensitive.

Um elemento, como dito anteriormente, pode possuir outros elementos como filhos. No caso do exemplo acima, podemos estender o elemento *livro* definindo os elementos filhos *capítulo*, como pode ser visto na Figura 3.6. A Figura 3.7 demonstra graficamente a árvore do elemento *livro*.

```
<livro autor="J. R. R. Tolkien" titulo="Senhor do Anéis"
      subtítulo="O Retorno do Rei">
  <capítulo numero="1" nome="Minas Tirith" início="5"/>
  <capítulo numero="4" nome="O cerco de Gondor" início="69"/>
</livro>
```

Figura 3.6: Representação em código do elemento *livro*.

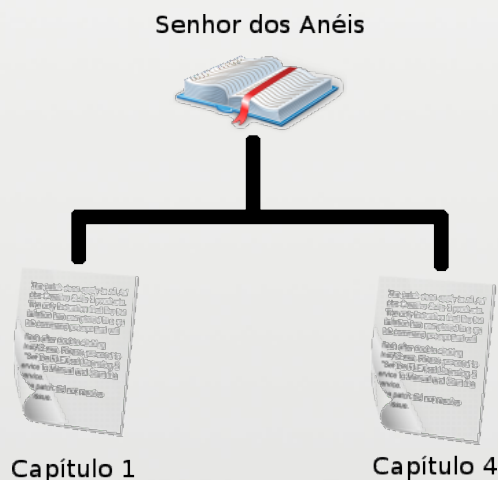


Figura 3.7: Representação gráfica do elemento *livro*.

Neste segundo exemplo, pode-se perceber uma segunda notação para um elemento. O elemento não é terminado por `</>`, como apresentado anteriormente, mas pela repetição do nome do elemento entre os símbolos `</>` e `>`, como visto no exemplo em `</livro>`. Outro ponto a ser observado é que os elementos *capítulo* filhos de *livro* são definidos dentro deste último, mantendo uma organização e hierarquia entre os elementos.

3.3 Estrutura de um documento NCL

Para possibilitar a criação de documentos hipermídia seguindo o modelo NCM apresentado anteriormente, foi desenvolvida a linguagem NCL. NCL é uma linguagem baseada em XML.

Todo o conteúdo de um documento NCL é definido dentro do elemento `<ncl>` sendo sua estrutura dividida em duas grandes partes, o cabeçalho e o corpo do documento.

3.3.1 Cabeçalho

No cabeçalho do documento NCL são definidas as características de apresentação do documento, através das regiões e descritores, os relacionamentos que serão utilizados para a sincronização dos nós, através dos conectores, entre outras facilidades.

3.3.2 Corpo

Uma vez tendo sido definido o cabeçalho do documento NCL, é necessária definição dos elementos e a sincronização entre eles. O corpo do documento NCL fica então responsável por esta definição. Além da definição dos nós e elos, o corpo de um documento NCL pode também definir composições, como visto na introdução sobre o modelo NCM, para melhorar sua estrutura e organização. Vale ressaltar que o corpo do documento NCL é considerado um tipo especial de composição e como toda composição deve possuir pelo menos uma porta para que seu conteúdo seja exibido.

Após esta introdução sobre a linguagem NCL, podemos passar para o aprendizado da linguagem. As seções seguintes apresentarão gradativamente as facilidades da linguagem NCL explicando os elementos que as proveem. Além das explicações serão propostos exercícios para a fixação dos conceitos apresentados e promover desde o princípio um maior contato com a linguagem. Ao final deste capítulo, seguindo os complementos gradativos dos exemplos, o leitor terá criado um programa em NCL que exibe um botão em quatro momentos diferentes e conta o número de vezes que esse botão é selecionado.

3.4 Definindo a apresentação

Nesta seção, definiremos como serão apresentados os elementos que compõem o programa sendo criado. Como visto anteriormente, essas definições são feitas através das regiões e descritores.

3.4.1 Regiões

As regiões indicam as áreas da tela onde os elementos do programa (vídeo, texto, imagem, etc) poderão ser apresentados. Com isso o documento NCL define a posição inicial de um elemento. Para que um documento seja apresentado, ao menos uma região deverá ser definida, sendo essa a região que definirá a dimensão da tela onde o documento NCL será apresentado.

As regiões são definidas no cabeçalho do documento NCL na base de regiões, elemento `<regionBase>` e são definidas pelo elemento `<region>`. Uma região pode também conter outras regiões para permitir uma maior organização do documento NCL. O exemplo da Figura 3.8 demonstra a criação de duas regiões (*rgFundo* e *rgVideo*).

```
<region id="rgFundo" width="640" height="480">  
  <region id="rgVideo" left="320" top="0"  
    width="320" height="240"/>  
</region>
```

Figura 3.8: Criação de duas regiões.

A região *rgFundo* define o tamanho total do dispositivo onde o documento será apresentado, neste caso o tamanho será de 640 por 480 pixels. A região *rgFundo* possui uma região interna *rgVideo*. Essa região define o local onde, provavelmente, um vídeo será apresentado, neste caso no canto superior direito da tela e tendo o tamanho de 1/4 do tamanho total da tela.

Pode-se observar que como a região *rgVideo* está contida na região *rgFundo*, seu posicionamento e tamanho (caso este fosse representado de forma percentual) tem como referência a região *rgFundo*.

Note que toda região possui um identificador único representado pelo atributo **id**. Este identificador será utilizado por outros elementos do documento NCL sempre que uma referência à região for necessária. Uma região pode possuir os atributos listados abaixo. Destes atributos, apenas o atributo **id** é obrigatório.

- **id**: identificador da região. Este valor, que deve ser único, será utilizado sempre que uma referência à região for necessária.
- **left**: Define a coordenada horizontal à esquerda da região. Esta coordenada pode ser indicada em valores absolutos (sem a necessidade de indicar a unidade de medida usada) ou percentuais. Este valor terá como referência a região pai ou, no caso de a região em questão não estar contida em outra, a tela do dispositivo de exibição.
- **top**: Define a coordenada vertical superior da região. Esta coordenada pode ser indicada em valores absolutos (sem a necessidade de indicar a unidade de medida usada) ou percentuais. Este valor terá como referência a região pai ou, no caso de a região em questão não estar contida em outra, a tela do dispositivo de exibição.
- **right**: Define a coordenada horizontal à direita da região. Esta coordenada pode ser indicada em valores absolutos (sem a necessidade de indicar a unidade de medida usada) ou percentuais. Este valor terá como referência a região pai ou, no caso de a região em questão não estar contida em outra, a tela do dispositivo de exibição.
- **bottom**: Define a coordenada vertical inferior da região. Esta coordenada pode ser indicada em valores absolutos (sem a necessidade de indicar a unidade de medida usada) ou percentuais. Este valor terá como referência a região pai ou, no caso de a região em questão não estar contida em outra, a tela do dispositivo de exibição.
- **width**: Define a dimensão horizontal da região. Vale observar que esta dimensão também poderia ser definida com o uso dos atributos **left** e **right**. A escolha de como definir a dimensão da região fica a cargo do programador. Entretanto, no caso dos atributos **left** e **width** serem definidos, o valor do atributo **right** não é considerado.
- **height**: Define a dimensão vertical da região. Vale observar que esta dimensão também poderia ser definida com o uso dos atributos **top** e **bottom**. A escolha de como definir a dimensão da região fica a cargo do programador. Entretanto, no caso dos atributos **top** e **height** serem definidos, o valor do atributo **bottom** não é considerado.
- **zIndex**: Define a sobreposição das camadas. De acordo com o valor contido neste atributo, uma região será apresentada “acima” de outras regiões com **zIndex** menor e “abaixo” de outras regiões com **zIndex** maior. Caso duas regiões com mesmo valor de **zIndex** sejam definidas, no caso de uma mídia ser apresentada em cada região, existem duas possibilidades de ordenação das regiões: a mídia apresentada por último será apresentada “acima” da anterior; caso as duas sejam apresentadas ao mesmo tempo, a ordem é escolhida pelo formatador (executor).

3.4.2 Descritores

Os descritores definem como os elementos serão apresentados nas áreas da tela. Para isso, um descritor deve indicar a região à qual está relacionado e definir especificidades de sua apresentação.

Os descritores são definidos no cabeçalho do documento NCL na base de descritores, elemento `<descriptorBase>` e definido pelo elemento `<descriptor>`. O exemplo da Figura 3.9 demonstra a criação de dois descritores (*dpFundo* e *dpVideo*).

```
<descriptor id="dpFundo" region="rgFundo"/>
<descriptor id="dpVideo" region="rgVideo"/>
```

Figura 3.9: Criação de dois descritores.

Note que um descritor também define um identificador único. Além disso, um descritor define a região à qual este está associado através do atributo **region**. Um descritor não é utilizado somente para relacionar um elemento com uma região da tela. Além disso, o descritor pode ser utilizado para definir como será feita a navegação pela tela, através dos botões do controle remoto, bem como mudar a forma como um elemento é apresentado, mudando seu tempo de exibição ou sua característica de transparência.

Um descritor possui os seguintes atributos:

- **id**: Identificador do descritor. Este atributo contém um valor único que será utilizado em referências ao descritor.
- **region**: Identifica a região à qual o descritor está relacionado. Caso o elemento não possua conteúdo visual, não é necessário definir uma região.
- **explicitDur**: Define a duração da apresentação do elemento associado ao descritor. Caso uma duração não seja especificada neste atributo, será considerada a duração padrão de cada elemento. Se o elemento for um texto ou imagem, o tempo padrão de apresentação é infinito. Nos casos onde os elementos possuem uma duração, esta será considerada sua duração padrão. O valor definido neste atributo deve estar em segundos (considerando a unidade "s").
- **focusIndex**: Define um índice a ser utilizado na navegação entre os elementos apresentados na tela. Se um descritor não definir um índice, o elemento a ele associado não receberá foco na navegação. No início da execução do documento, o foco é passado para o elemento associado ao descritor de menor índice. Uma observação a ser feita é que o valor do índice pode não ser numérico, nesse caso será escolhido o índice lexicograficamente menor.
- **moveLeft**: Define o descritor, através do seu índice, que receberá foco quando o botão "seta para esquerda" do controle remoto for pressionado. Esse mapeamento só será realizado quando o descritor que o define estiver com o foco.
- **moveRight**: Define o descritor, através do seu índice, que receberá foco quando o botão "seta para direita" do controle remoto for pressionado. Esse mapeamento só será realizado quando o descritor que o define estiver com o foco.
- **moveUp**: Define o descritor, através do seu índice, que receberá foco quando o botão "seta para cima" do controle remoto for pressionado. Esse mapeamento só será realizado quando o descritor que o define estiver com o foco.

- **moveDown:** Define o descritor, através do seu índice, que receberá foco quando o botão "seta para baixo" do controle remoto for pressionado. Esse mapeamento só será realizado quando o descritor que o define estiver com o foco.
- **focusBorderColor:** Define a cor da borda da região associada ao descritor quando o elemento utilizando o descritor recebe foco. O atributo pode assumir um dos seguintes valores: white, black, silver, gray, red, maroon, fuchsia, purple, lime, green, yellow, olive, blue, navy, aqua, ou teal.
- **focusBorderWidth:** Define a espessura da borda apresentada em pixels. A espessura pode assumir valores positivos e negativos. No caso de um valor positivo ser atribuído, a borda será apresentada fora da região, caso negativo, a borda será apresentada dentro da região.

3.4.3 Prática

Agora que aprendemos como criar regiões e descritores, podemos começar a criação da nossa aplicação NCL. Começaremos definindo as regiões e descritores onde serão apresentados os seguintes elementos:

- vídeo: vídeo principal apresentado durante todo o programa;
- botão: botão apresentado em alguns momentos.

A Figura 3.10 demonstra a criação da base de regiões, enquanto a Figura 3.11 demonstra a criação da base de descritores.

```
<regionBase>
  <region id="rg_video" left="0" top="0" width="100%"
    height="100%" zIndex="1">
    <region id="rg_button" left="40%" top="40%" width="20%"
      height="20%" zIndex="3"/>
  </region>
</regionBase>
```

Figura 3.10: Base de regiões.

```
<descriptorBase>
  <descriptor id="dp_video" region="rg_video"/>
  <descriptor id="dp_button" region="rg_button" focusIndex="0"/>
</descriptorBase>
```

Figura 3.11: Base de descritores.

3.5 Inserindo os elementos

Para que um documento NCL tenha algo a apresentar, é necessária a definição dos conteúdos que serão apresentados. Estes conteúdos são definidos pelas mídias de NCL como veremos a seguir. Os elementos definidos nesta seção pertencem ao corpo do documento NCL.

Tabela 3.1: MIME Types utilizados por NCL.

MIME Type	Significado
text/html	.htm, .html
text/css	.css
text/xml	.xml
image/bmp	.bmp
image/png	.png
image/gif	.gif
image/jpeg	.jpg
audio/basic	.wav
audio/mp3	.mp3
audio/mp2	.mp2
audio/mpeg4	.mp4, .mpg4
video/mpeg	.mpeg, .mpg
application/x-ginga-NCLua	.lua

3.5.1 Mídias

Uma mídia, ou seja, a representação do objeto que será apresentado pelo documento, é definida pelo elemento `<media>` de NCL.

Na Figura 3.12 podemos ver como é feita a criação de um elemento `<media>`.

```
<media id="video" src="video.avi" descriptor="dpVideo"/>
```

Figura 3.12: Criação de um elemento media.

Uma mídia possui os seguintes atributos:

- **id**: Identificador da mídia será utilizado, assim como os demais identificadores, em referências a esta mídia.
- **src**: Define a localização do conteúdo objeto de mídia representado (URI).
- **descriptor**: Define o descritor que controla a apresentação da mídia. O valor desse atributo deverá ser o identificador do descritor.
- **type**: Define o tipo da mídia (audio, video, text, etc). O valor desse atributo será um dos MIME Types definidos pela IANA (Internet Assigned Numbers Authority) [?].

A Tabela 3.1 demonstra os MIME Types utilizados por NCL.

3.5.2 Âncoras

As âncoras definem uma parte do conteúdo de uma mídia. Por parte do conteúdo de uma mídia, pode-se entender como um subconjunto dos elementos que compõem uma mídia. Por exemplo, digamos que queremos definir âncoras de um vídeo. Nesse caso o vídeo em si seria a mídia enquanto as âncoras seriam trechos desse vídeo.

As âncoras são utilizadas para permitir a sincronização de objetos de mídia (músicas, vídeos, textos, etc) com outros objetos de mídia. Imagine o exemplo de um filme legendado. Este filme

possui diversas cenas, sendo que cada cena possui um diálogo. Neste caso o filme deverá possuir uma âncora para cada cena, possibilitando assim que as legendas (diálogos) sejam apresentadas no tempo correto.

Uma âncora é definida pelo elemento `<area>`, sendo este um elemento filho de uma mídia. O exemplo da Figura 3.13 demonstra a criação de um vídeo com âncoras.

```
<media id="video" src="video.avi" descriptor="dpVideo">  
  <area id="cena01" begin="2s" end="5s"/>  
  <area id="cena02" begin="5s" end="6s"/>  
</media>
```

Figura 3.13: Criação de um vídeo com âncoras.

Uma âncora possui os seguintes atributos:

- **id**: Identificador único da âncora. Este atributo é usado em referências feitas a este elemento.
- **begin**: Início da âncora. Este atributo é usado quando é definida uma âncora de um objeto de mídia contínuo. O valor desse atributo pode ser no formato "segundo.fração" ou "hora:minuto:segundo.fração".
- **end**: Fim da âncora. Este atributo é usado em conjunto com o atributo **begin**, de forma a definir o final da âncora. Seu valor possui o mesmo formato do atributo **begin**. Caso este atributo (**end**) não seja definido, o final da âncora é considerado como sendo o final da mídia que a contém.
- **label**: Define um identificador para a âncora. Um **label** é usado em âncoras de objetos procedurais, podendo ser usado para identificar um conjunto de âncoras, como será visto mas adiante.

Uma âncora pode possuir outros atributos a serem usados com outros tipos de mídias, como mídias não contínuas, por exemplo. Por ser esta uma apostila básica de NCL, esses atributos não serão apresentados neste texto.

3.5.3 Propriedades

As propriedades definem, como o próprio nome indica, uma propriedade de uma mídia. Uma propriedade é usada quando a manipulação de algum atributo de uma mídia é necessária.

Uma propriedade é definida pelo elemento `<property>` sendo este um elemento filho da mídia. O elemento `<property>` contém apenas os atributos **name** e **value**.

O atributo **name** indica o nome da propriedade da mídia que será manipulada. O atributo **value**, por sua vez, indica o valor desta propriedade. O exemplo da Figura 3.14 demonstra o uso de propriedades.

Note que uma propriedade nem sempre é definida com um valor. Outro ponto a se notar é que, no caso de uma propriedade ser definida com um valor, este será o valor inicial do referido atributo.

```
<media id="video" src="video.avi" descriptor="dpVideo"/>
  <property name="descriptor"/>
  <property name="explicitDur" value="60s"/>
</media>
```

Figura 3.14: Uso de propriedades.

3.5.4 Prática

Agora que aprendemos como criar mídias, suas âncoras e propriedades, podemos continuar a criação da nossa aplicação NCL. Precisamos definir os seguintes elementos:

- vídeo: vídeo principal apresentado durante todo o programa;
- botão: botão apresentado em alguns momentos;
- contador: responsável por armazenar o número de vezes que o botão foi selecionado.

A Figura 3.15 demonstra a criação das mídias.

```
<media id="video" src="video.mpg" type="video/mpeg" descriptor="dp_video">
<area id="area01" begin="3s" end="6s"/>
<area id="area02" begin="10s" end="13s"/>
<area id="area03" begin="17s" end="20s"/>
<area id="area04" begin="24s" end="27s"/>
</media>

<media id="button" descriptor="dp_button" src="button.jpg" type="image/jpeg"/>

<media id="clicks" src="clicks.lua">
  <property name="inc"/>
  <property name="counter"/>
</media>
```

Figura 3.15: Mídias do documento.

3.6 Organizando o documento

Como visto na seção 3.1.1, o modelo NCM define, além de nós de conteúdo, nós de composição. Estes nós são usados para organizar e estruturar um documento seguindo o modelo. Nesta seção veremos como estruturar um documento NCL usando composições.

3.6.1 Contexto

Em NCL, um nó de composição é representado pelo elemento `<context>`. Este elemento pode ter outros elementos, inclusive outros contextos, como elementos filhos.

Um contexto permanece ativo enquanto um de seus elementos filhos estiver ativo e, caso uma interface específica do contexto não seja especificada em um elo, um contexto inicia todos

os seus elementos filhos quanto ele é iniciado. A criação de interfaces para contextos será vista mais a frente.

Um contexto define também um atributo **id** para identificá-lo em relacionamentos que venha a participar. O exemplo da Figura 3.16 demonstra a criação de um contexto.

```
<context id="context01">  
  ... elementos ...  
</context>
```

Figura 3.16: Criação de um contexto.

3.6.2 Portas

Uma porta define uma interface para um contexto. Como o modelo NCM não permite que um elemento dentro de um contexto seja acessado diretamente de fora do contexto, é necessário o uso de portas sempre que tal ação seja necessária.

Uma porta contém os seguintes atributos:

- **id**: Identificador único permitindo que esta seja referenciada por outro elemento do documento NCL.
- **component**: nó componente sendo mapeado pela porta.
- **interface**: Interface do nó componente sendo mapeado. Podem ser **âncoras** ou **propriedades**, caso o elemento mapeado seja uma **mídia**, ou outras **portas**, caso o elemento mapeado seja um **contexto**.

O corpo do documento NCL, elemento **<body>**, é considerado um tipo especial de contexto. Logo o corpo do documento NCL deve conter pelo menos uma porta, mapeando um nó definido dentro dele, para que o documento possa ser apresentado.

3.6.3 Prática

Agora que aprendemos como estruturar nosso documento, podemos continuar com nosso aplicativo. Precisamos definir os seguintes elementos:

- corpo do documento ncl;
- uma porta para o primeiro elemento a ser apresentado.

A Figura 3.17 demonstra a criação dos contextos e suas portas. Os elementos apresentados são os mesmos criados na Seção 3.5.4.

3.7 Sincronizando os elementos

Até agora aprendemos como criar elementos, organizar o documento NCL e definir quais elementos serão exibidos no início do documento. Além disso, precisamos definir como os elementos se relacionam durante a apresentação e ao receber a interação do usuário. Os conectores e links são utilizados para esse propósito.

```
<body>
  <port component="video" id="inicio"/>

  <media id="video" src="video.mpg" type="video/mpeg" descriptor="dp_video">
    <area id="area01" begin="3s" end="6s"/>
    <area id="area02" begin="10s" end="13s"/>
    <area id="area03" begin="17s" end="20s"/>
    <area id="area04" begin="24s" end="27s"/>
  </media>

  <media id="button" descriptor="dp_button" src="button.jpg" type="image/jpeg"/>

  <media id="clicks" src="clicks.lua">
    <property name="inc"/>
    <property name="counter"/>
  </media>
</body>
```

Figura 3.17: Corpo do documento e contexto da tela.

3.7.1 Conectores

Os conectores definem relações genéricas que serão utilizadas pelos elementos de um documento NCL. Durante essa definição não são indicados os participantes de um relacionamento específico.

Imagine, por exemplo, a relação "ensina à". Esta relação define que alguém ensina a alguém, porém não define que ensina e quem aprende, ou seja, os participantes.

Um conector é definido numa base de conectores, elemento `<connectorBase>` pelo elemento `<causalConnector>`. Este elemento define uma relação de causa e efeito, como seu próprio nome já indica, através de papéis de condição e papéis de ação. Um papel pode ser compreendido como uma interface de um conector e é este que indica qual a participação que um nó tem na relação.

Voltando ao exemplo citado anteriormente, a relação "ensina à" possui dois papéis, o de quem ensina e o de quem aprende. O exemplo da Figura 3.18 demonstra a criação de um conector.

```
<causalConnector id="onBeginStart">
  <simpleCondition role="onBegin"/>
  <simpleAction role="start" max="unbounded" qualifier="par"/>
</causalConnector>
```

Figura 3.18: Criação de um conector.

Pode-se reparar no exemplo acima que a condição é dada pelo papel "onBegin", indicando que a condição espera pelo início da apresentação de um elemento, enquanto a ação é dada pelo papel "start", indicando que a apresentação de um elemento será iniciada.

Condições Simples

O elemento `<simpleCondition>` define uma condição que deverá ser satisfeita para que o conector seja ativado. Este elemento define, através do atributo **role** o nome do papel de condição. NCL possui um conjunto de nomes reservados para papéis de condição. Estes nomes identificam uma condição, ou ação, sem a necessidade de mais informações. Os nomes e seus significados são listados abaixo:

- **onBegin**: É ativado quando a apresentação do elemento ligado a esse papel é iniciada.

- **onEnd:** É ativado quando a apresentação do elemento ligado a esse papel é terminada.
- **onAbort:** É ativado quando a apresentação do elemento ligado a esse papel é abortado.
- **onPause:** É ativado quando a apresentação do elemento ligado a esse papel é pausada.
- **onResume:** É ativado quando a apresentação do elemento ligado a esse papel é retomada após um pause.
- **onSelection:** É ativado quando uma tecla a ser especificada for pressionada durante a apresentação do elemento ligado a esse papel ou quando a tecla ENTER for pressionada enquanto o elemento estiver com foco.
- **onBeginAttribution:** É ativado logo antes de um valor ser atribuído a uma propriedade do elemento ligado a esse papel.
- **onEndAttribution:** É ativado logo após um valor ser atribuído a uma propriedade do elemento ligado a esse papel.

Quando a condição "onSelection" é utilizada, torna-se necessário definir o atributo **key** do elemento `<simpleCondition>`. Este valor indica qual a tecla que deverá ser pressionada para que o conector seja ativado. Os valores possíveis são: "0" a "9", "A" a "Z", "*", "#", "MENU", "INFO", "GUIDE", "CURSOR_DOWN", "CURSOR_LEFT", "CURSOR_RIGHT", "CURSOR_UP", "CHANNEL_DOWN", "CHANNEL_UP", "VOLUME_DOWN", "VOLUME_UP", "ENTER", "RED", "GREEN", "YELLOW", "BLUE", "BACK", "EXIT", "POWER", "REWIND", "STOP", "EJECT", "PLAY", "RECORD" e "PAUSE". Este valor pode ser estabelecido com o uso de parâmetros do conector como será visto mais adiante.

O elemento `<simpleCondition>` pode definir outros tipos de condições diferentes das condições padrão apresentadas acima. Esta facilidade, entretanto, não será apresentada neste texto.

Condições Compostas

Além de uma condição simples, um conector pode definir uma condição composta. Uma condição composta é definida pelo elemento `<compoundCondition>`. Este elemento possui duas ou mais condições simples como filhas. Quando utilizado, este elemento deve definir um atributo **operator** que recebe os valores "and" ou "or", indicando se todas ou pelo menos uma condição deve ser satisfeita para que o conector seja ativado.

Ações Simples

O elemento `<simpleAction>` define uma ação a ser executada quando o conector é ativado. Este elemento define, através do atributo **role** o nome do papel de ação. Além do nome do papel, este elemento define através do atributo **max** o número máximo de elementos que poderão utilizar este papel. O valor "unbounded" define um número máximo ilimitado. Caso o atributo **max** seja definido, é necessária a definição de outro atributo chamado **qualifier**, este define se as ações serão executadas em paralelo ou sequencialmente, valores "par" e "seq" respectivamente.

NCL também possui um conjunto de nomes reservados para papéis de ação. Os nomes e seus significados são listados abaixo:

- **start:** Inicia a apresentação do elemento ligado a esse papel.
- **stop:** Termina a apresentação do elemento ligado a esse papel.
- **abort:** Aborta a apresentação do elemento ligado a esse papel.

- **pause:** Pausa a apresentação do elemento ligado a esse papel.
- **resume:** Retoma a apresentação do elemento ligado a esse papel.
- **set:** Estabelece um valor a uma propriedade de um elemento associado a esse papel.

Quando o papel "set" é usado, a condição que o define deve também declarar o atributo **value**. Este atributo é responsável por indicar o valor a ser recebido pela propriedade. Este valor pode ser estabelecido com o uso de parâmetros do conector como será visto mais adiante.

O elemento `<simpleAction>` pode definir outros tipos de ações diferentes das ações padrão apresentadas acima. Esta facilidade, entretanto, não será apresentada neste texto.

Ações compostas

Além de uma ação simples, um conector pode definir uma ação composta. Uma ação composta é definida pelo elemento `<compoundAction>`. Uma ação composta possui outras ações simples como filhas. Quando utilizado, este elemento deve definir um atributo **operator** que recebe os valores "par" ou "seq", indicando se as ações deverão ser executadas em paralelo ou sequencialmente.

Parâmetros

Um conector também pode definir parâmetros. Esses parâmetros são definidos pelo elemento `<connectorParam>` e utilizados para que o valor a ser avaliado, ou estabelecido, possa ser indicado no momento do uso do conector. O exemplo da Figura 3.19 ilustra esse conceito.

```
<causalConnector id="onKeySelectionStart">
  <connectorParam name="keyCode"/>
  <simpleCondition role="onSelection" key="$keyCode"/>
  <simpleAction role="start" max="unbounded" qualifier="par"/>
</causalConnector>
```

Figura 3.19: Definição de parâmetro em conector.

Note que o parâmetro define apenas o seu nome, deixando a definição do seu valor para o momento do seu uso. Os locais onde o valor desse parâmetro for utilizado indicam esta funcionalidade através do valor "\$nome_do_parâmetro".

3.7.2 Elos

Um elo é utilizado para identificar os elementos participantes de uma relação. Seguindo o exemplo de relação "ensina à" apresentada na seção anterior, um elo que utilizasse essa relação identificaria os elementos "professor" e "aluno". O relacionamento completo, especificado pelo elo, ficaria então "o professor ensina ao aluno".

Um elo é definido pelo elemento `<link>`. Este elemento define a relação sendo utilizada através do seu atributo **xconnector**.

Para a criação de ligações entre os elementos e os papéis, um elo define elementos filhos `<bind>`. Um `<bind>` define os atributos:

- **role:** Identifica o papel sendo utilizado.

- **component:** Identifica, através do seu id, o elemento participando da relação.
- **interface:** Identifica uma âncora ou propriedade do elemento, caso este seja uma mídia, ou uma porta de um elemento, caso este seja uma composição, através do seu id.

O exemplo da Figura 3.20 demonstra a criação de um elo.

```
<link id="link1" xconnector="onKeySelectionStart">  
  <bind component="video" role="onSelection">  
    <bindParam name="keyCode" value="YELLOW"/>  
  </bind>  
  <bind component="bkg" role="start"/>  
  <bind component="screen" role="start"/>  
</link>
```

Figura 3.20: Criação de elo.

No exemplo acima podemos notar a existência do elemento `<bindParam>`. Este elemento é utilizado em casos onde a passagem de parâmetros é necessária. Este elemento define os atributos **name** e **value**, onde o primeiro identifica o parâmetro e o segundo o seu valor.

3.7.3 Prática

Agora que aprendemos como criar relacionamentos em nosso documento, podemos continuar a criação do nosso programa. Precisamos definir as seguintes relações:

- quando um elemento começar, começar outro elemento;
- quando um elemento terminar, terminar outro elemento;
- quando um elemento for selecionado, definir o valor de um outro elemento.

e os elos que utilizam essas relações (conectores).

A Figura 3.21 demonstra a criação dos conectores que serão usados pelo documento NCL. Enquanto a Figura 3.22 demonstra a utilização dos mesmos.

```
<causalConnector id="onBeginStart">
  <simpleCondition role="onBegin"/>
  <simpleAction role="start"/>
</causalConnector>

<causalConnector id="onEndStop">
  <simpleCondition role="onEnd"/>
  <simpleAction role="stop" max="unbounded"/>
</causalConnector>

<causalConnector id="onSelectionSet">
  <simpleCondition role="onSelection"/>
  <connectorParam name="var"/>
  <simpleAction role="set" value="$var"/>
</causalConnector>
```

Figura 3.21: Conectores da base de conectores.

```
<link xconnector="onBeginStart">
  <bind role="onBegin" component="video"/>
  <bind role="start" component="clicks"/>
</link>

<link xconnector="onBeginStart">
  <bind role="onBegin" component="video" interface="area01"/>
  <bind role="start" component="button"/>
</link>
<link xconnector="onBeginStart">
  <bind role="onBegin" component="video" interface="area02"/>
  <bind role="start" component="button"/>
</link>
<link xconnector="onBeginStart">
  <bind role="onBegin" component="video" interface="area03"/>
  <bind role="start" component="button"/>
</link>
<link xconnector="onBeginStart">
  <bind role="onBegin" component="video" interface="area04"/>
  <bind role="start" component="button"/>
</link>

<link xconnector="onEndStop">
  <bind role="onEnd" component="video" interface="area01"/>
  <bind role="stop" component="button"/>
</link>
<link xconnector="onEndStop">
  <bind role="onEnd" component="video" interface="area02"/>
  <bind role="stop" component="button"/>
</link>
<link xconnector="onEndStop">
  <bind role="onEnd" component="video" interface="area03"/>
  <bind role="stop" component="button"/>
</link>
<link xconnector="onEndStop">
  <bind role="onEnd" component="video" interface="area04"/>
  <bind role="stop" component="button"/>
</link>

<link xconnector="onSelectionStopSet">
  <bind role="onSelection" component="button"/>
  <bind role="set" component="clicks" interface="inc">
    <bindParam name="var" value="1"/>
  </bind>
</link>
```

Figura 3.22: Links utilizados no programa.

Tabela 3.2: Comparadores utilizados em regras NCL.

eq	igual a
ne	diferente de
gt	maior que
ge	maior ou igual a
lt	menor que
le	menor ou igual a

3.8 Definindo alternativas

Até agora, vimos como criar um programa NCL, definindo como seus elementos serão apresentados, que elementos serão apresentados e a ordem em que serão apresentados. A este tipo de ordem de apresentação podemos chamar de fluxo temporal do programa NCL.

Além das facilidades apresentadas, NCL também permite que um programa possa ter seu fluxo temporal modificado durante sua apresentação. Essa modificação é feita através da definição de alternativas de conteúdo. Uma alternativa define, para um determinado momento da apresentação do programa NCL, possíveis caminhos a serem seguidos. A escolha de um caminho é dada pela avaliação de condições, chamadas regras.

3.8.1 Regras

As regras representam condições que podem ser usadas em um documento NCL. Uma regra, basicamente, compara uma propriedade de um elemento NCL com um valor, retornando o resultado dessa comparação. Uma regra é definida pelo elemento `<rule>`.

Uma regra define os seguintes atributos:

- **id**: Identificador da regra, é referenciado quando o uso da mesma for necessário;
- **var**: Indica que propriedade estará sendo testada. O valor desse atributo será o identificador de uma **propriedade**;
- **comparator**: Indica que tipo de comparação será feita pela regra;
- **value**: Indica o valor que será comparado com a propriedade.

A Tabela 3.2 demonstra os possíveis comparadores utilizados em uma regra.

O exemplo da Figura 3.23 demonstra a criação de uma regra.

```
<rule id="rule_pt" var="idioma" comparator="eq" value="pt"/>
```

Figura 3.23: Criação de uma regra.

Uma regra é definida na base de regras, representada pelo elemento `<ruleBase>`, no cabeçalho do documento NCL.

3.8.2 Switch

Um switch é um elemento que apresenta as alternativas de conteúdo que um programa NCL pode tomar em um determinado momento. Este elemento avalia uma série de regras e, dependendo da regra avaliada verdadeira, ativa um dos elementos definidos dentro dele.

Um switch é ativado da mesma forma como um elemento qualquer de NCL, seu identificador pode ser utilizado em elos tornando possível que este seja ativado, parado ou mesmo participe de alguma condição de disparo. O exemplo da Figura 3.24 demonstra a criação de um switch.

```
<switch id="escolhe_audio">
  <bindRule rule="rule_en" constituent="audio_en"/>
  <bindRule rule="rule_pt" constituent="audio_pt"/>

  <media id="audio_en" ... />
  <media id="audio_pt" ... />
</switch>
```

Figura 3.24: Criação de um switch.

Para que um switch escolha um elemento para ser apresentado, é necessária a definição de mapeamentos. Esses mapeamentos são definidos pelo elemento `<bindRule>` como visto no exemplo acima. Um mapeamento define a regra que deve ser avaliada e, caso esta seja verdadeira, o elemento que deverá ser ativado.

Vale ressaltar, que as regras são avaliadas de forma sequencial, sendo a **primeira** regra avaliada como verdadeira a que irá definir qual elemento será apresentado. Com isso, mesmo que mais de uma regra seja verdadeira em um determinado momento, somente um elemento será exibido.

Caso um switch possua elementos do tipo context, é necessário também se indicar a porta de destino de cada mapeamento. Essa indicação é feita pelo elemento `<switchPort>` como mostrado no exemplo da Figura 3.25.

```
<switch id="escolhe_audio">
  <switchPort id="mapeamento">
    <mapping component="context_en" interface="port_en" />
    <mapping component="context_pt" interface="port_pt" />
  </switchPort>

  <bindRule rule="rule_en" constituent="context_en" />
  <bindRule rule="rule_pt" constituent="context_pt" />

  <context id="context_en">
    <port id="port_en" ... />
    ...
  </context>

  <context id="context_pt">
    <port id="port_pt" ... />
    ...
  </context>
</switch>
```

Figura 3.25: Porta em switch.

Referências

- [1] Soares, L.F.G., Barbosa, S.D.J. Programando em NCL: desenvolvimento de aplicações para middleware Ginga, TV digital e Web. Editora Elsevier. Rio de Janeiro, 2009. ISBN 978-85-352-3457-2.
- [2] W3C World-Wide Web Consortium. XML Schema Part 0: Primer Second Edition. W3C Recommendation. 2004. (<http://www.w3.org/TR/xmlschema-0/>).
- [3] Soares L.F.G, Rodrigues R.F. Nested Context Model 3.0: Part 1 – NCM Core. Technical Report, Departamento de Informática PUC-Rio. 2005.
- [4] W3C World-Wide Web Consortium. (<http://www.w3.org/>)

Capítulo 4

Linguagem Lua

4.1 Introdução

4.1.1 Pequeno histórico

A linguagem Lua [1] foi criada em 1993 dentro do Tecgraf [2] na PUC-Rio pelos professores Roberto Ierusalimschy, Luiz Henrique de Figueiredo e Waldemar Celes.

Um dos grandes parceiros do Tecgraf era a Petrobras, que demandava por programas para engenharia com interação gráfica. Em 1993 foram desenvolvidas duas linguagens para atender essa demanda, estas linguagens foram batizadas de DEL, coordenada pelo professor Luiz, e SOL, coordenada pelo professor Roberto.

Em meados de 1993, os professores Roberto, Luiz e Waldemar se reuniram para discutir SOL e DEL e concluíram que as duas linguagens poderiam ser substituídas por apenas uma. Assim nasceu a equipe dos desenvolvedores de Lua.

Lua foi parcialmente inspirada em SOL, por isso um amigo do Tecgraf, Carlos Henrique Levy, sugeriu o nome 'Lua'. Assim nasceu Lua.

4.1.2 Convenções Léxicas

Na linguagem Lua, nomes - ou identificadores - podem ser qualquer cadeia de letras, dígitos e sublinhados, porém não podem começar com um dígito. Os identificadores são utilizados para nomear funções, variáveis e campos de tabelas.

Algumas palavras são reservadas e não podem ser utilizadas como nomes. A Figura 4.1 mostra a lista das palavras reservadas pela linguagem.

and	break	do	else	elseif	
end	false	for	function	if	
in	local	nil	not	or	
repeat	return	then	true	until	while

Figura 4.1: Palavras reservadas.

Lua diferencia maiúsculas de minúsculas. Portanto for será interpretado diferente de For ou fOr ou foR ou FOR ou FOr ou FoR.

Uma *string* (cadeia de caracteres) pode ser delimitada com o uso de aspas simples ou aspas duplas e podem conter sequências de escape no estilo de C. A Figura 4.2 exhibe uma lista com os caracteres de escape:

```
'\a' (campainha)
'\b' (backspace)
'\f' (alimentação de formulário)
'\n' (quebra de linha)
'\r' (retorno de carro)
'\t' (tabulação horizontal)
'\v' (tabulação vertical)
'\' ' (barra invertida)
'\ "' (citação [aspa dupla])
'\ ' ' (apóstrofo [aspa simples])
```

Figura 4.2: Caracteres de escape.

4.1.3 O interpretador de linha de comando

Apesar de Lua ter sido originalmente projetada como uma linguagem de extensão, ela é frequentemente utilizada como linguagem principal. Um interpretador para Lua, chamado *lua*, é disponibilizado com a distribuição padrão da linguagem. O interpretador inclui todas as bibliotecas padrão.

A sintaxe para uso é: `lua [options] [script [args]]`.

As opções são:

- `-e stat`: executa a cadeia *stat*;
- `-l mod`: "requisita" *mod*;
- `-i`: entra em modo interativo após executar *script*;
- `-v`: imprime informação de versão;
- `-`: pára de tratar opções;
- `-:` executa *stdin* como um arquivo e pára de tratar opções.

Após tratar as opções passadas, *lua* executa o *script* fornecido, passando para ele os argumentos. Quando é chamado sem argumentos, *lua* comporta-se como `lua -v -i`, entrando no modo interativo. Na Figura 4.3 é possível observar a tela do interpretador *lua* em modo interativo.

Em sistemas Unix, *scripts* Lua podem ser usados como programas executáveis. Para isso basta inserir na primeira linha do *script* uma chamada com o caminho para o interpretador *lua*, como pode-se observar na Figura 4.4.

Para tornar o *script* executável: `chmod +x nome-do-script.lua`

4.1.4 Prática

Vamos iniciar nossa aventura pelo mundo da Lua.

Primeiro iremos instalar o interpretador Lua. Se você já tem o interpretador instalado em seu computador, sinta-se livre para pular esta parte.

Alguns sistemas Unix-like disponibilizam pacotes pré-compilados do interpretador. Em sistemas Debian-like, com os repositórios atualizados, digite: `aptitude search lua` para verificar se existe um pacote do interpretador Lua disponível.

Para instalar: `aptitude install nome-do-pacote-lua`.

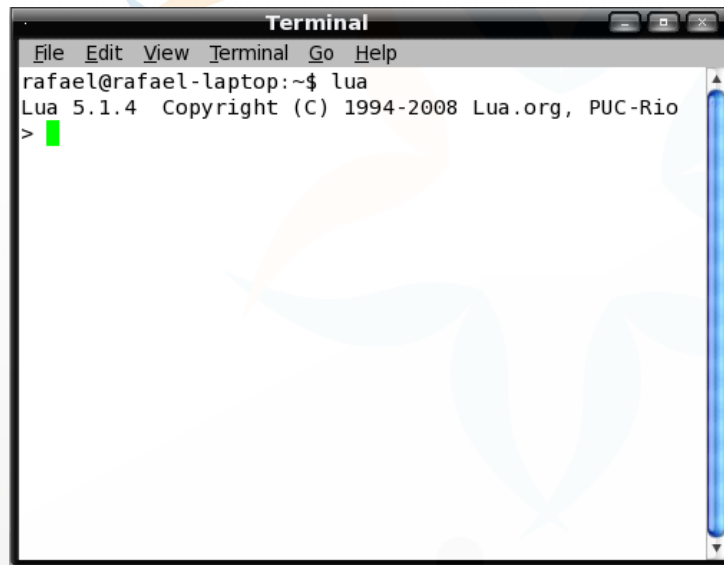


Figura 4.3: Interpretador Lua em modo interativo.

```
#!/usr/local/bin/lua
```

Figura 4.4: Chamada com caminho do interpretador Lua.

Para instalar no Slackware, entre no site <http://www.linuxpackages.net> e procure pelo pacote mais adequado para sua versão. Você pode baixar a versão mais nova diretamente deste link:

<http://mirror.slackwarebrasil.org/linuxpackages//Slackware/Slackware-12.1/Library/Lua/lua-5.1.4-i486-1gds.tgz>

Para instalar, logado como root, digite: **installpkg nome-do-pacote-lua.tgz**.

Após instalar o interpretador Lua, abra o terminal e digite: **lua**.

Se tudo ocorreu bem, você entrará no modo interativo do interpretador. A tela será semelhante a da Figura 4.3.

Usuários do Microsoft Windows, não desanimem! Existe esperança para os que ainda não alcançaram a luz! O interpretador Lua também foi portado para esse sistema. Maiores informações podem ser encontradas neste site: <http://luaforwindows.luaforge.net>.

4.2 Tipos e variáveis

Para uma melhor compreensão desta seção, segue uma definição formal para o termo *variável*:

Variável é um objeto (uma posição, frequentemente localizada na memória) capaz de reter e representar um valor ou expressão. Enquanto as variáveis só "existem" em tempo de execução, elas são associadas a "nomes", chamados identificadores, durante o tempo de desenvolvimento.

Quando nos referimos à variável, do ponto de vista da programação de computadores, estamos tratando de uma "região de memória (do computador) previamente identificada cuja finalidade é armazenar os dados ou informações de um programa por um determinado espaço de tempo".

4.2.1 Tipos

Lua é uma linguagem de tipagem dinâmica. Isso significa que não é necessário declarar o tipo de uma variável; ele é determinado dinamicamente dependendo do valor que a variável está armazenando.

Existem oito tipos básicos em Lua: *nil*, *boolean*, *number*, *string*, *userdata*, *function*, *thread*, e *table*. Observe a Figura 4.5.

```
print(type("Hello world")) --> string
print(type(10.4*3))         --> number
print(type(print))          --> function
print(type(type))           --> function
print(type(true))           --> boolean
print(type(nil))            --> nil
print(type(type(X)))        --> string
```

Figura 4.5: Tipos.

A função *print* imprime, a princípio, um conjunto de caracteres na tela.

A função *type* retorna uma *string* com o nome do tipo do valor recebido. Repare que, na última linha, a saída da função *print* é uma *string*; isso não é causado pelo valor de "X", mas sim porque a função *type* retorna uma *string*.

- *nil*: é utilizado para indicar a ausência de um valor; se uma variável for criada e nenhum valor for atribuído, seu valor será igual a *nil*, por exemplo;
- *boolean*: representa os valores booleanos *true* e *false*. Se uma expressão for igual a *nil* ou *false* será considerada falsa, sendo verdadeira caso contrário (inclusive quando ela for 0);
- *number*: Lua possui apenas um tipo numérico básico, *number*, que é ponto flutuante por padrão;
- *string*: representa cadeias de caracteres, que podem ser delimitadas por aspas duplas ou simples. Se desejar escrever uma cadeia que se estende por várias linhas, deve-se usar a notação `[[` para abrir a cadeia e `]]` para fechar a cadeia;
- *userdata*: é usado para armazenar dados C em variáveis Lua. Valores deste tipo somente podem ser criados ou modificados através da API de Lua com C. Nesta apostila não abordaremos a API de Lua com C, se desejar obter maiores informações sobre esse assunto: *google is your friend*;
- *function*: tipo que representa funções. Em Lua, funções são valores de primeira classe, isso significa que um valor do tipo função não requer nenhum tratamento especial. Assim, a mesma variável pode ser utilizada para armazenar um valor numérico ou uma função;
- *thread*: representam fluxos de execução independentes. O tipo *thread* dá suporte ao uso de co-rotinas. O uso de co-rotinas não será abordado nesta versão da apostila;
- *table*: representa uma tabela Lua. Tabelas são o único mecanismo de estruturação de dados de Lua. Com tabelas podemos representar arrays, tabelas de símbolos, conjuntos, grafos, registros, etc;

4.2.2 Variáveis

Para declarar uma variável basta escrever seu nome. Em Lua os nomes de variáveis devem começar com letras ou sublinhado.

Para atribuir um valor a uma variável, utiliza-se o operador de atribuição `=`. Operadores serão melhor estudados na seção 4.3. Um pequeno exemplo pode ser visto na Figura 4.6.

```
a = 10           -- a é declarada e recebe o valor 10.
b, c = "oi", 20  -- b recebe a string "oi" e c recebe 20.
d               -- o valor de d é nil
print(b,d)       --> oi    nil
```

Figura 4.6: Exemplo de atribuição.

Numa operação de atribuição, os valores do lado direito são avaliados e, posteriormente, atribuídos às variáveis do lado esquerdo. É possível efetuar atribuição de mais de um valor para mais de uma variável com um único comando, como visto na Figura 4.7.

```
a, b = "olá", "lua"
print(a, b)      --> oi   lua
```

Figura 4.7: Mais um exemplo de atribuição.

Se existirem mais valores do que variáveis, os valores extra são descartados. Caso contrário, as variáveis extras recebem **nil**.

```
a, b = "olá", "mundo", "da lua"
print(a, b)      --> olá  mundo

a, b, c = "olá", "mundo"
print(a, b, c)   --> olá  mundo  nil
```

Em Lua as variáveis, por padrão, são consideradas globais. Para restringirmos o escopo de uma variável precisamos utilizar a palavra chave **local**, sendo que o escopo de uma variável local é determinado pelo bloco de código onde ela foi declarada.

Se uma variável é declarada como local fora de qualquer bloco de código, ela será visível em todo o arquivo. Blocos de código serão abordados na seção 4.4.

4.2.3 Prática

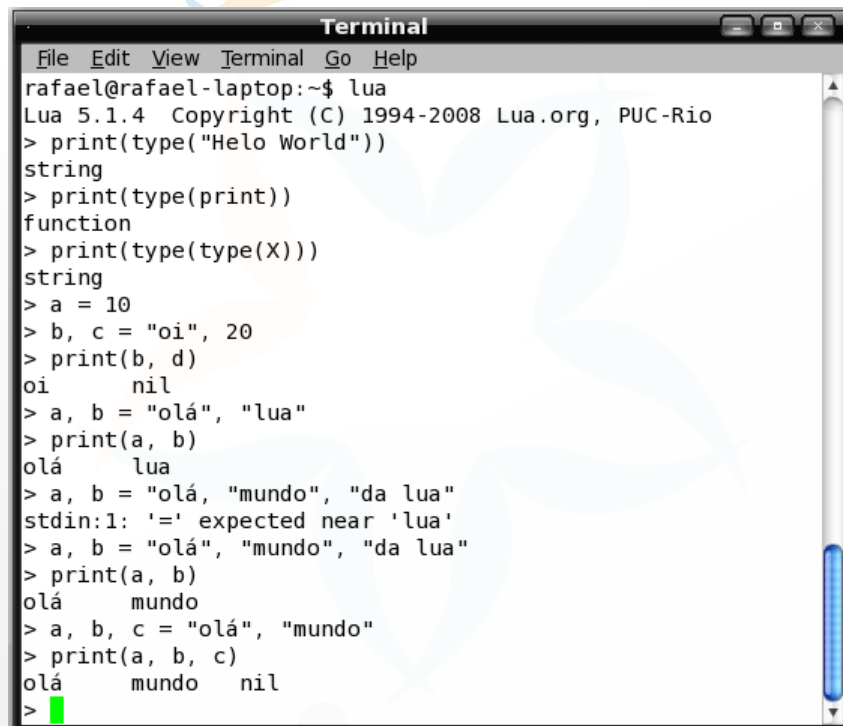
Vamos trabalhar um pouco nossos conhecimentos sobre variáveis.

Abra o terminal e execute o interpretador lua em modo interativo. Em seguida execute alguns exemplos que foram dados nesta seção, sua saída deverá ser semelhante a da Figura 4.8.

Reparem na mensagem de erro: `stdin:1: '=' expected near 'lua'`.

Ela é causada, propositalmente, por não fecharmos, com aspas dupla, a declaração da *string* `olá`.

Ao apertar o cursor para cima o interpretador lua exibe os comandos que foram digitados, assim como o *shell* do Linux. Portanto para corrigir este erro, basta retornar para o comando que o causou e fechar a declaração da *string*.



```

Terminal
File Edit View Terminal Go Help
rafael@rafael-laptop:~$ lua
Lua 5.1.4 Copyright (C) 1994-2008 Lua.org, PUC-Rio
> print(type("Helo World"))
string
> print(type(print))
function
> print(type(type(X)))
string
> a = 10
> b, c = "oi", 20
> print(b, d)
oi      nil
> a, b = "olá", "lua"
> print(a, b)
olá     lua
> a, b = "olá", "mundo", "da lua"
stdin:1: '=' expected near 'lua'
> a, b = "olá", "mundo", "da lua"
> print(a, b)
olá     mundo
> a, b, c = "olá", "mundo"
> print(a, b, c)
olá     mundo  nil
>

```

Figura 4.8: Saída esperada para os exemplos.

4.3 Operadores

4.3.1 Aritiméticos

Lua oferece suporte para os operadores aritméticos mais utilizados:

- + Adição;
- - subtração;
- * multiplicação;
- / divisão;
- ^ exponenciação;
- - negação.

Na Figura 4.9 é possível observar o uso de alguns destes operadores.

Se deseja obter maiores informações sobre operações matemáticas com Lua, consulte o manual de referência em:

<http://www.lua.org/manual/5.1/pt/>

4.3.2 Relacionais

Lua suporta os seguintes operadores relacionais:

- < Menor que;

```
a = 5 + 2
print(a)    --> 7

a = 5 - 2
print(a)    --> 3

b = 3 * 2
print(b)    --> 6

b = 6 / 2
print(b)    --> 3

c = 2^(2)
print(c)    --> 4

d = 4 * -3
print(d)    --> -12
```

Figura 4.9: Operadores.

- > maior que;
- <= menor ou igual a;
- >= maior ou igual a;
- == igual a;
- ~= negação da igualdade (diferente).

Os operadores relacionais sempre retornam **true** ou **false**.

Lua compara os tipos *table*, *userdata* e *function* por referência. Dois valores serão considerados iguais apenas se forem exatamente o mesmo objeto.

Após a execução deste código:

```
a = {}; a.x = 1; a.y = 0
b = {}; b.x = 1; b.y = 0
c = a
```

Temos *a* **igual** a *c*, porém **diferente** de *b*.

4.3.3 Lógicos

Os operadores lógicos são:

- **and**;
- **or**;
- **not**.

Em Lua os operadores lógicos consideram **false** e **nil** como falso e qualquer outra coisa como verdadeiro.

O operador **and** retorna o primeiro argumento se ele for falso, caso contrário retorna o segundo. O operador **or** retorna o primeiro argumento se ele não for falso, caso contrário retorna o segundo argumento. Os dois operadores só avaliam o segundo argumento quando necessário.

Para um melhor entendimento observe o exemplo abaixo:

```
print(2 and 3)      --> 3
print(nil and 1)    --> nil
print(false and 1)  --> false
print(2 or 3)       --> 3
print(false or 3)   --> 3
```

O operador **not** retorna **true** ou **false**:

```
print(not nil)      --> true
print(not false)    --> true
print(not 0)        --> false
print(not not nil)  --> false
```

4.3.4 Concatenação

Lua permite concatenação de *strings* e para isso disponibiliza o operador **..** (dois pontos seguidos). Se na operação de concatenação um dos operandos for um número, este será convertido para *string*.

A operação de concatenação cria uma nova *string*, sem alterar os operandos originais.

Exemplo:

```
print("Olá " .. "mundo " .. "da Lua.") --> Olá mundo da Lua.
a = 0 .. 1
print(type(a))                          --> string

a = "Olá"
print(a .. " mundo da Lua.")            --> Olá mundo da Lua.
print(a)                                --> Olá
```

4.3.5 Prática

Para fixação dos conceitos apresentados nesta seção iremos escrever um código Lua para calcular a média ponderada entre dois valores.

Vamos construir nosso primeiro *script* Lua. Escolha seu editor de textos preferido (sugestões: **gedit** ou **Vim**) e digite o código apresentado na Figura 4.10.

Agora salve com um nome específico (por exemplo: *script.lua*).

Torne seu *script* executável: **chmod +x script.lua**

Para ver o resultado entre no diretório onde salvou seu *script* e digite: **./script.lua**.

```
1  #!/usr/bin/lua
2
3  print("\n\n")
4
5  print("Digite o primeiro valor:")
6  valor1 = io.stdin:read'*l'
7
8  print("Digite o peso:")
9  peso_valor1 = io.stdin:read'*l'
10
11 print("Digite o segundo valor:")
12 valor2 = io.stdin:read'*l'
13
14 print("Digite o peso:")
15 peso_valor2 = io.stdin:read'*l'
16
17 media = ((valor1 * peso_valor1) + (valor2 * peso_valor2))
18 media = media / (peso_valor1 + peso_valor2)
19
20 print("A media ponderada eh: " .. media .. "\n\n")
```

Figura 4.10: Mão na massa.

Repare que na **linha 6** utilizamos uma função da biblioteca de E/S padrão de Lua. Para maiores detalhes consulte o manual de referência [3].

Outra observação importante é que nosso script não valida os dados digitados, logo um usuário desavisado poderia não digitar um número ou mesmo digitar uma *string*, isso faria seu programa "quebrar".

4.4 Estruturas de controle

Em Lua pode-se utilizar um conjunto, não tão extenso porém suficiente, de estruturas de controle. **if** e/ou **else** para condicionais e **while**, **repeat** e **for** para interação.

Todo bloco de código de uma estrutura de controle deve ter um terminador explícito. Para **if**, **for**, e **while** o terminador é o **end**; para o **repeat** o terminador é o **until**.

4.4.1 if then else

A estrutura **if** testa se uma condição é verdadeira ou falsa. Se a condição for verdadeira a parte *then* é executada, senão *else* é executada. A parte *else* é opcional.

Veja alguns exemplos na Figura 4.11.

```
if a < 6 then a = 0 end

if a < b then return a else return b end

if line > MAXLINES then
    showpage()
    line = 0
end
```

Figura 4.11: Exemplo de estrutura de controle.

Em Lua não existe a estrutura *switch-case*. Se você deseja construir uma estrutura de **ifs** aninhados poderá utilizar **elseif** que é similar a um **else** seguido por um **if**. Veja alguns exemplos:

```
if casa == "azul" then
  dono = "jose"
elseif casa == "verde" then
  dono = "manoel"
elseif casa == "amarela" then
  dono = "maria"
elseif casa == "vermelha" then
  dono = "madalena"
else
  error("Casa sem dono")
end
```

4.4.2 while

O **while** é uma das estruturas de repetição de Lua. Primeiramente a condição é testada, se for falsa não entra no *loop*; caso contrário, Lua executa o bloco do *loop* e avalia novamente a condição.

Segue um exemplo:

```
local i = 1
while a[i] do
  print(a[i])
  i = i + 1
end
```

4.4.3 repeat

A estrutura **repeat-until** repete o que está em seu bloco de código até a condição (testada pelo **until**) ser verdadeira. O exemplo abaixo escreve na tela a primeira linha não-vazia.

```
repeat
  line = os.read()
until line ~= ""
print(line)
```

Repare que a condição só é testada no final da estrutura, portanto o bloco de código é sempre executado ao menos uma vez.

4.4.4 for numérico

Essa estrutura poderia ser conhecida como o **for** tradicional, quem já programou em C ou Java, por exemplo, observará que sua sintaxe é bem familiar.

```
for var=exp1,exp2,exp3 do
    bloco de código
end
```

Temos que *exp1* é o valor inicial de *var*. O *bloco de código* será executado enquanto *var* for menor ou igual *exp2*, no caso de um passo positivo, ou maior ou igual a *exp2*, no caso de um passo negativo. A *exp3* representa o passo do laço (valor incrementado), quando não é especificada assume-se que seu valor é 1. Observe alguns exemplos:

```
for i=1,f(x) do print(i) end

for i=10,1,-1 do print(i) end
```

A variável *var* (no exemplo acima *i*) é visível apenas no corpo da estrutura e não deve ser alterada. Um erro típico é presumir que a variável *var* existirá depois que o *loop* terminar.

```
for i=1,100 do print(i) end
test = i      -- errado! 'i' aqui é global
```

A construção correta deveria ser como a apresentada na Figura 4.12.

```
-- encontrar um valor em uma lista
local found = nil
for i=1,a.n do
    if a[i] == value then
        found = i      -- salvar o índice do valor ('i')
        break
    end
end
print(found)
```

Figura 4.12: for numérico: construção correta.

4.4.5 for genérico

O *for* genérico é utilizado para percorrer os valores retornados por uma função iteradora. Os principais iteradores oferecidos por Lua são:

- *pairs*: percorre as chaves de uma tabela;
- *ipairs*: percorre os índices de um *array*;
- *io.lines*: percorre as linhas de um arquivo;
- *string.gfind*: percorre palavras de uma *string*;

Ainda é possível escrevermos nossos próprios iteradores, porém isso não será abordado nesta apostila.

No exemplo da Figura 4.13 *a* é um *array*. Para imprimirmos todos os seus valores podemos utilizar um iterador.

```
for i, v in ipairs (a) do    -- i armazena o índice, v o valor
    print (v)
end
```

Figura 4.13: for genérico: exemplo.

Para imprimir todas as chaves de uma tabela *t* (em qualquer ordem), podemos proceder como exibido na Figura 4.14.

```
for k, v in pairs (t) do    -- k armazena a chave, v o valor
    print (k, v)
end
```

Figura 4.14: for genérico: mais um exemplo.

4.5 Saindo de um bloco

Para sair de um bloco de programas podemos utilizar os comandos **break** e **return**. O comando **break** é muito utilizado em estruturas de controle, pois permite terminar a execução da estrutura.

Com o comando **return** é possível retornar valores de uma função ou simplesmente terminar sua execução. Funções serão abordadas na Seção 4.6.

Na utilização destes comandos deve-se atentar para o fato de que eles devem ser os últimos comandos do bloco de código. Observe o exemplo da Figura 4.15.

```
i = 5
while i < 10 do
    break          -- Erro!
    i = i + 1
end
```

Figura 4.15: Exemplo de saída de bloco.

O código da Figura 4.15 não será executado corretamente. Existem duas formas para solucionar este problema. Podemos reordenar o código, para deixar o comando **break** ou **return** como último comando do bloco, ou criar um novo bloco **do-end** ao redor do comando. Na Figura 4.16 pode-se observar um exemplo com a segunda abordagem.

4.6 Funções

Segue uma pequena definição.

Em ciência da computação, mais especificamente no contexto da programação, uma função (subrotina, procedimento ou mesmo subprograma) consiste numa porção


```
i = 5
while i < 10 do
  do
    break    -- Ok! break como último comando do bloco
  end
  i = i + 1  -- Atribuição como último comando do bloco 'while'
end
```

Figura 4.16: Correção do exemplo de saída de bloco.

de código que resolve um problema muito específico, parte de um problema maior (a aplicação final). [4]

Em Lua as funções são o principal mecanismo para abstração e expressão. Podem executar tanto uma tarefa (conhecido como procedimento ou subrotina) como computar e retornar valores.

4.6.1 Declaração e execução

Uma função pode ser declarada da forma mostrada na Figura 4.17.

```
function nome_da_funcao (arg_1, arg_2, ..., arg_n)
  corpo_da_funcao
end
```

Figura 4.17: Declaração de função.

As variáveis *arg-N* são os parâmetros da função. Em Lua uma função pode ter múltiplos argumentos, assim como pode não ter nenhum.

Como exemplo (Figura 4.18), vamos definir uma função que recebe dois parâmetros e retorna a soma deles.

```
function soma (a, b)
  local x = a or 1    -- x = 1 se a é falso
  local y = b or 1    -- y = 1 se b é falso
  return x + y
end
```

Figura 4.18: Funções: exemplo 1.

Repare que existe uma falha nesta função. Sabe a resposta? Continue no caminho da luz...

A função soma possui dois parâmetros, porém é possível chamá-la com um número diferente de parâmetros. Se chamarmos a função soma com mais parâmetros do que ele espera, os valores extras serão descartados. Se a função for chamada com um número menor de parâmetros, o valor dos parâmetros que não foram fornecidos será **nil**. Observe o exemplo da Figura 4.19.

4.6.2 Número variável de argumentos

Lua permite a declaração de funções com um número variável de argumentos. Para isso basta utilizar três pontos (...) como argumento da função.

```
s = soma(2, 3)  -- a = 2, b = 3
print(s)        --> 5

s = soma(2)     -- a = 2, b = nil, y = 1
print(s)        --> 3
```

Figura 4.19: Funções: exemplo 2.

Para acessar os argumentos que foram passados, utilizamos a notação ..., para criar um *array* onde o primeiro argumento da função está na posição 1, o segundo na posição 2 e assim consecutivamente. Observe o exemplo (Figura 4.20) que mostra uma função que recebe uma *string* e imprime aquelas com tamanho maior que 4.

```
function maior_que_4 (...)
  for i, t in ipairs {...} do -- percorre a lista de argumentos
    if #t > 4 then            -- # é o operador de tamanho para cadeia de arrays
      print (t)
    end
  end
end
maior_que_4 ("abobora", "sol", "lua", "balao") -- Imprime "abobora" e "balao"
```

Figura 4.20: Funções: exemplo 3.

4.6.3 Retorno

Uma função pode não retornar nenhum valor, retornar apenas um valor ou retornar múltiplos valores. Contudo, nem sempre é possível obter todos os valores retornados por uma função.

Para um melhor entendimento, observe a função da Figura 4.21.

```
function soma_multiplica (a, b)
  local x = a or 1  -- x = 1 se a é falso
  local y = b or 1  -- y = 1 se b é falso
  return x + y, x * y
end
```

Figura 4.21: Exemplo de retorno em função.

4.6.4 Prática

Vamos criar uma função para testar se uma variável é do tipo *string*.

Se o valor recebido for do tipo desejado ela deve retornar **true**, senão retorna **false**.

A definição da função pode ser vista na Figura 4.22.

Essa função pode ser utilizada em outra parte de seu programa, como mostrado na figura 4.23.

```
function verifica_string(var)
    if (type(var) == "string") then
        return true
    else
        return false
    end
end
```

Figura 4.22: Prática: definição da função.

```
...

if(verifica_string(teste)) then
    print "A variavel teste eh do tipo string"
else
    print "A variavel teste NAO eh do tipo string"
end

...
```

Figura 4.23: Prática: programa utilizando função.

4.7 Tabelas

Em Lua, tabelas são o único mecanismo de estrutura de dados existente. Elas implementam *arrays*, matrizes, conjuntos e diversas outras estruturas. Nesta apostila não exploraremos toda a potencialidade das tabelas. Para maiores detalhes, consulte o manual [3].

4.7.1 Criação de tabelas

Tabelas são criadas utilizando o construtor `{}` e podem ser criadas de várias formas diferentes.

Para criar uma tabela vazia basta fazer: `tabela = {}`.

Pode-se também criar a tabela já iniciando alguns valores, como pode ser visto na Figura 4.24.

```
tabela = {"semana","engenharia",2009}
```

Figura 4.24: Criando tabela com valores.

Com a execução do código da Figura 4.24, `tabela[1]` terá a *string* "semana", `tabela[2]` a *string* "engenharia" e `tabela[3]` o valor 2009.

Uma outra forma de inicializar tabelas é como *array*, neste caso devemos informar o valor e a chave. A Figura 4.25 exemplifica este uso.

```
tabela = {p="semana",e="engenharia",t=2009,a="UFF"}
print(tabela["e"], tabela.t) --> engenharia 2009
```

Figura 4.25: Criando tabela como *array*.

Após a execução do código apresentado na Figura 4.25, a tabela *tabela* possui quatro chaves. Para inserir um novo valor nesta tabela, basta fazer uma nova atribuição: `tabela.cinco = "peta5"`.

Ainda podemos inserir valores em uma tabela utilizando a função `table.insert`. Para maiores detalhes sobre a função `table.insert` consulte o manual [3].

Uma tabela também pode conter outras tabelas. A Figura 4.26 exemplifica este uso.

```
tabela = {peta={cinco=5}}  
print (tabela.peta.cinco) --> 5
```

Figura 4.26: Criando tabela que contém outra tabela.

4.7.2 Tamanho de tabelas

É possível saber o tamanho de uma tabela, que está sendo utilizada como *array*, utilizando o operador `#`.

Este operador retorna um índice *n* do *array* onde o valor da posição *n* é diferente de `nil` e o valor da posição *n + 1* é `nil`. Evite o uso deste operador em *arrays* que possam conter valores `nil` no meio deles.

É possível ver um exemplo de uso do operador `#` na Figura 4.27.

```
tabela = {p="semana",e="engenharia",t=2009,a="UFF"}  
print(#tabela)      --> 4  
table.insert(tabela,"cinco")  
print(#tabela)      --> 5
```

Figura 4.27: Exemplo de uso do operador `#`.

4.7.3 Prática

Nosso objetivo nesta prática é criar uma função que testa alguns valores de uma determinada tabela. Isso será muito utilizado no restante da apostila.

Primeiro vamos criar nossa tabela (Figura 4.28).

```
event = {}  
event.post = {  
    class = 'ncl',  
    type = 'attribution',  
    property = 'myProp',  
    action = 'start',  
    value = '10'  
}
```

Figura 4.28: Prática: criação da tabela.

Agora criaremos a função que receberá nossa tabela e testará os valores de alguns campos. Pode ser visto na Figura 4.29.

```
function handler(evt)
  if evt.class ~= 'ncl' then return end
  if evt.type ~= 'attribution' then return end
  if evt.property ~= 'myProp' then return end

  print("Acao: ", evt.action)
  print("Valor: ", evt.value)

  return
end
```

Figura 4.29: Prática: criação da função que testará a tabela.

Na função apresentada na Figura 4.29 os campos *class*, *type* e *property* são testados. Se não contiverem os valores desejados, a função é finalizada. Se os valores desejados forem encontrados, imprime o conteúdo dos campos *action* e *value*.

A Figura 4.30 exibe o código completo.

```
--tabela
event = {}
event.post = {
  class = 'ncl',
  type = 'attribution',
  property = 'myProp',
  action = 'start',
  value = '10'
}

--funcao verificadora
function handler(evt)
  if evt.class ~= 'ncl' then return end
  if evt.type ~= 'attribution' then return end
  if evt.property ~= 'myProp' then return end

  print("Acao: ", evt.action)
  print("Valor: ", evt.value)

  return
end

--uso da funcao
handler(event) --> start    10
```

Figura 4.30: Prática: código completo.

O código da Figura 4.30 pode parecer não ter muito sentido neste momento. Nas próximas seções ele ficará mais claro.

Referências

- [1] R. Ierusalimschy, L. H. de Figueiredo, W. Celes. The evolution of Lua. Proceedings of ACM HOPL III (2007) 2-1–2-26.
- [2] Grupo de Tecnologia em Computação Gráfica. <http://www.tecgraf.puc-rio.br/>
- [3] Manual de Referência de Lua 5.1. <http://www.lua.org/manual/5.1/pt/>
- [4] Wikipédia. <http://pt.wikipedia.org/wiki/Subrotina>

Capítulo 5

Integração NCL-Lua

5.1 Introdução

A importância da integração entre as linguagens NCL e Lua [1] se deve principalmente ao fato de NCL se tratar de uma linguagem declarativa, pecando, portanto, pela falta de um maior controle das informações processadas, o que já não acontece com uma linguagem procedural como Lua. Por exemplo, não é de fácil implementação permitir a entrada de dados por parte do telespectador utilizando apenas NCL. Este capítulo descreve como é feita essa integração, a qual permite um aumento do potencial dos programas para a televisão digital.

5.2 Scripts NCLua

Os scripts NCLua usam a mesma abstração para objetos de mídia utilizada para imagens, vídeos e outras mídias. Em outras palavras, um documento NCL apenas relaciona objetos de mídia, não importando qual o tipo de seu conteúdo. Esses scripts seguem sintaxe idêntica à de Lua e possuem um conjunto de bibliotecas similar. Ou seja, a linguagem Lua possui adaptações para funcionar embutida em documentos NCL. O arquivo Lua deve ser escrito em um arquivo (com extensão .lua) separado do documento NCL, que apenas o referencia, como qualquer outro nó de mídia.

A principal peculiaridade do NCLua está no fato de que o seu ciclo de vida é controlado pelo documento NCL que o referencia, ou seja, fica definido no documento NCL em que instante o programa Lua se inicia. As funções podem receber um valor de tempo como um parâmetro opcional que pode ser usado para especificar o exato momento que o comando deve ser executado.

O modelo de execução de um nó NCLua é orientado a eventos, isto é, o fluxo da aplicação é guiado por eventos externos oriundos do formatador NCL. O acionamento desses eventos ora é feito pelo formatador NCL, ora é feito pelo NCLua que sinaliza ao formatador uma mudança interna. Esse comportamento caracteriza uma ponte de comunicação bidirecional entre o formatador e o NCLua.

5.3 Módulos NCLua

As bibliotecas disponíveis para o NCLua estão divididas em quatro módulos essenciais onde cada um exporta um conjunto de funções. Esses módulos são:

- **Módulo event:** Permite a comunicação do NCLua com o formatador NCL através de eventos.

- **Módulo canvas:** Oferece uma API para desenhar primitivas gráficas e imagens.
- **Módulo settings:** Exporta a tabela settings com variáveis definidas pelo autor do documento NCL e variáveis de ambiente reservadas, contidas no nó application/x-ginga-settings.
- **Módulo persistent:** Oferece uma API para exportar a tabela persistent com variáveis definidas em uma área reservada, restrita do middleware.

5.3.1 Módulo event

O módulo event é o módulo mais importante por fazer a ponte entre o documento NCL e o script NCLua. Para que o NCLua receba um evento do formatador, o script deve registrar pelo menos uma função de tratamento de eventos. A partir de então, qualquer ação tomada pela aplicação é apenas em resposta ao evento recebido.

A função event.register é usada conforme mostrado na Figura 5.1.

```
function handler (evt)
...    -- código para tratar os eventos
end

event.register(handler)    -- registro do tratador
```

Figura 5.1: Uso da função event.register.

Um NCLua também pode enviar eventos para se comunicar com o ambiente. Para isso, utiliza-se a função event.post, como exibido na Figura 5.2.

```
event.post (dst; evt)
```

Figura 5.2: Uso da função event.post.

Os eventos são tabelas Lua simples sendo o campo *class* responsável por identificar a classe do evento. Podemos definir cinco classes distintas:

- **ncl:** Um NCLua se comunica com o documento no qual está inserido através desta classe de eventos.

Por exemplo, consideremos que o NCLua recebeu um evento conforme a Figura 5.3.

```
{class='ncl', type='presentation', action='start'}
```

Figura 5.3: NCLua recebendo evento.

Isso indica que a mídia .lua irá iniciar sua apresentação. E para o NCLua postar um evento ao formator, ele pode, por exemplo, proceder conforme exibido na Figura 5.4.

Isso indica que a mídia .lua irá parar sua apresentação.

Os campos *type* e *action* serão explicados mais adiante.

- **key:** utilizada para representar o uso do controle remoto pelo usuário.

```
event.post {class='ncl', type='presentation', action='stop'}
```

Figura 5.4: NCLua postando evento.

- **tcp:** O uso do canal de interatividade é realizado por meio desta classe de eventos.
- **sms:** envio e recebimento de mensagens de texto.
- **user:** permite a criação de eventos próprios. Como eventos desta classe são para uso interno, não faz sentido a postagem de seus eventos com o destino igual a 'out'.

Na norma da ABNT [2], pode-se encontrar outros tipos de classes, no entanto estas não foram aqui mencionadas por ainda não estarem implementadas na última versão do Ginga.

Para a classe de eventos *ncl*, utilizamos os seguintes campos:

- **type:** Pode ser do tipo *presentation*, do tipo *attribution* ou do tipo *selection*.
- **action:** Pode assumir os seguintes valores: *start*, *stop*, *pause*, *resume* ou *abort*.
- **area** ou **property:** Eventos podem ser direcionados a âncoras específicas ou ao nó como um todo. Quando ausente, assume o nó inteiro.
- **value:** Recebe o valor da propriedade definida no campo *property*, no caso onde *type* é do tipo *attribution*.

A Figura 5.5 mostra um modelo:

```
event.post {  
  class    = 'ncl',  
  type     = 'attribution',  
  property = 'myProp',  
  action   = 'start',  
  value    = '10',  
}
```

Figura 5.5: NCLua postando evento de atribuição.

Para a classe de eventos *key*, utilizamos os seguintes campos:

- **type:** Deve ser do tipo *press* ou do tipo *release*.
- **key:** Valor da tecla em questão.

Para a classe de eventos *user* nenhum campo da tabela representando o evento está definido (além, claro, do campo *class*).

Funções do módulo event

- **event.register (pos, f, class):** registra a função passada como um tratador de eventos.
pos: [number] Posição de inserção da função de tratamento (opcional).
f: [função] Função de tratamento.
class: [string] Filtro de classe (opcional).
O parâmetro *pos* indica a posição em que *f* é registrada. Caso não seja passado, a função registrada será a última a receber eventos.
A função *f* é a função de tratamento de eventos.
O parâmetro *class* indica que classe de eventos a função deve receber.
A assinatura de *f* deve ser como mostrado na Figura 5.6.

```
function f (evt)
  -- returns boolean
end
```

Figura 5.6: Função de tratamento de eventos.

Onde *evt* é o evento que, ao ocorrer, ativa a função *f*.

- **event.unregister (f):** cancela o registro da função passada como um tratador de eventos.
f: [função] Função de tratamento.
Novos eventos não serão mais passados a *f*.
- **event.post (dst, evt):** posta (gera) um evento.
Onde: dst: [string] É o destino do evento.
evt: [table] Evento a ser postado.
O parâmetro *dst* pode assumir os valores 'in' (envio para a própria aplicação) e 'out' (envio para o formatador). Caso seja omitido, assume o valor 'out'.
Retorna:
sent: [boolean] Se o evento foi enviado com sucesso.
err_msg: [string] Mensagem em caso de erro.
- **event.timer (time, f):** cria um timer de *time* milissegundos e, ao fim deste, chama a função *f*.
Onde:
time: [number] Tempo em milissegundos.
f: [função] Função de retomada.
Retorna:
unreg: [função] Função que, quando chamada, cancela o timer.
A assinatura de *f* deve ser:
function f () end
O valor de zero milissegundos é válido.

- **event.uptimer (time)**: retorna o número de milissegundos decorridos desde o início da aplicação.

Onde:

time: [number] Tempo em milissegundos.

5.3.2 Módulo canvas

Um NCLua tem a possibilidade de fazer operações gráficas durante a apresentação de uma aplicação, tais como desenho de linhas, círculos, imagens, etc. Tudo isso é possível, pois o módulo canvas oferece uma API gráfica para ser usada por aplicações NCLua.

Quando um objeto de mídia NCLua é iniciado, a região do elemento `<media>` correspondente (do tipo `application/x-ginga-NCLua`) fica disponível como a variável global `canvas` para o script Lua. Se o elemento de `<media>` não tiver nenhuma região especificada (propriedades `left`, `right`, `top` and `bottom`), então o valor para `canvas` deve ser estabelecido como `"nil"`.

Funções do módulo canvas

- **canvas:new(width, height)**: criação de novo objeto gráfico com tamanho especificado.

Onde:

width: [number] Largura do canvas.

height: [number] Altura do canvas.

- **canvas:new(image_path)**: uma imagem é passada como parâmetro.

Onde:

image_path: [string] Caminho da imagem.

- **canvas:attrSize()**: retorna as dimensões do canvas.

Retorna `width` e `height`, nessa ordem. Não é possível alterar as dimensões de um canvas instanciado.

- **canvas:attrColor(R, G, B ,A)**: altera a cor do canvas.

As cores são passadas em RGB.

O argumento `A` varia de 0 (totalmente transparente) à 255 (totalmente opaco).

As primitivas são desenhadas com a cor desse atributo e o seu valor inicial é `'0, 0, 0, 255'`.

Também é possível passar diretamente o nome da cor: `canvas:attrColor(color_name, A)`.

Onde `'color_name'` pode assumir os seguintes valores: `'white'`, `'aqua'`, `'lime'`, `'yellow'`, `'red'`, `'fuchsia'`, `'purple'`, `'maroon'`, `'blue'`, `'navy'`, `'teal'`, `'green'`, `'olive'`, `'silver'`, `'gray'` e `'black'`.

- **canvas:attrColor()**: retorna as cores do canvas.

Retorna `R`, `G`, `B` e `A`, nessa ordem.

- **canvas:attrClip(x, y, width, height)**: limita a área do canvas para desenho.

Onde:

x: [number] Coordenada x da área limitada.

y: [number] Coordenada y da área limitada.

width: [number] Largura da área limitada.

height: [number] Altura da área limitada.

O valor inicial é o canvas inteiro.

- **canvas:attrClip()**: retorna as dimensões da área limitada.

Retorna na ordem: x, y, width e height.

- **canvas:attrCrop(x, y, width, height)**: atributo de corte do canvas.

Onde:

x: [number] Coordenada x da área limitada.

y: [number] Coordenada y da área limitada.

width: [number] Largura da área limitada.

height: [number] Altura da área limitada.

Quando o canvas é composto sobre outro, apenas a região de recorte é copiada para o canvas de destino. O valor inicial é o canvas inteiro.

- **canvas:attrCrop()**: retorna as dimensões de corte do canvas.

Retorna na ordem: x, y, width e height.

- **canvas:attrFont(face, size, style)**: altera a fonte do canvas.

Onde:

face: [string] Nome da fonte size: [number] Tamanho da fonte style: [string] Estilo da fonte

As seguintes fontes devem obrigatoriamente estar disponíveis: 'Tiresias', 'Verdana'.

O tamanho é em pixels e representa a altura máxima de uma linha escrita com a fonte escolhida.

Os estilos possíveis são: 'bold', 'italic' ou 'bold-italic'.

O valor nil assume que nenhum dos estilos será usado.

- **canvas:attrFont()**: retorna a fonte do canvas.

Retorna na ordem: face, size e style.

- **canvas:attrFlip (horiz, vert)**: espelhamento do canvas usado em funções de composição.

Onde:

horiz: [boolean] Se o espelhamento for horizontal.

vert: [boolean] Se o espelhamento for vertical.

- **canvas:attrFlip (horiz, vert)**: retorna a configuração de espelhamento do canvas.

Retorna *horiz* se for horizontal e *vert* se for vertical.

- **canvas:attrOpacity (opacity)**: altera a opacidade do canvas.

Onde:

opacity: [number] Novo valor de opacidade do canvas.

- **canvas:attrOpacity ()**: retorna a opacidade do canvas.
- **canvas:attrRotation (degrees)**: configura o atributo de rotação do canvas.
Onde:
degree: [number] Rotação do canvas em graus (deve ser múltiplo de 90).
- **canvas:attrRotation (degrees)**: retorna o atributo de rotação do canvas.
- **canvas:attrScale (w, h)**: escalona o canvas com nova largura e altura.
Onde:
w: [number] Largura de escalonamento do canvas.
h: [number] Altura de escalonamento do canvas.
Um dos valores pode ser true, indicando que a proporção do canvas deve ser mantida.
O atributo de escalonamento é independente do atributo de tamanho, ou seja, o tamanho do canvas é mantido.
- **canvas:attrScale ()**: retorna os valores de escalonamento do canvas.
Retorna na ordem: w e h.
- **canvas:drawLine (x1, y1, x2, y2)**: desenha uma linha.
A linha tem extremidades em (x1,y1) e (x2,y2).
Utiliza a cor especificada em attrColor.
- **canvas:drawRect (mode, x, y, width, height)**: desenha um retângulo.
Onde:
mode: [string] Modo de desenho
x: [número] Coordenada do retângulo
y: [número] Coordenada do retângulo
width: [número] Largura do retângulo
height: [número] Altura do retângulo
O parâmetro *mode* pode receber *frame* para desenhar a moldura ou *fill* para preenchê-lo.
- **canvas:drawRoundRect(mode, x, y, width, height, arcWidth, arcHeight)**: desenha um retângulo com bordas arredondadas.
Onde:
arcWidth: [número] Largura do arco do canto arredondado
arcHeight: [número] Altura do arco do canto arredondado
Os outros parâmetros funcionam da mesma forma que em canvas:drawRect.
- **canvas:drawPolygon(mode)**: desenha um polinômio.
O parâmetro *mode* pode receber:
open: para desenhar o polígono sem ligar o último ponto ao primeiro;
close: para desenhar o polígono ligando o último ponto ao primeiro;
fill: para desenhar o polígono ligando o último ponto ao primeiro e colorir a região interior.

- **canvas:drawEllipse (mode, xc, yc, width, height, ang_start, ang_end):** desenha uma elipse.

Onde:

mode: [string] Modo de desenho

xc: [número] Centro da elipse

yc: [número] Centro da elipse

width: [número] Largura da elipse

height: [número] Altura da elipse

ang_start: [número] Ângulo de início

ang_end: [número] Ângulo de fim

O parâmetro *mode* pode receber *arc* para desenhar apenas a circunferência ou *fill* para preenchimento interno.

- **canvas:drawText (x, y, text):** desenha um texto.

Onde:

x: [número] Coordenada do texto

y: [número] Coordenada do texto

text: [string] Texto a ser desenhado

Desenha o texto utilizando a fonte configurada em `canvas:attrFont()`.

- **canvas:clear (x, y, w, h):** limpa o canvas com a cor configurada em `attrColor`.

Onde:

x: [número] Coordenada da área de clear

y: [número] Coordenada da área de clear

w: [número] Largura da área de clear

h: [número] Altura da área de clear

Caso não sejam passados os parâmetros de área, assume-se que o canvas inteiro será limpo.

- **canvas:flush ():** atualiza o canvas após operações de desenho e de composição.

É suficiente chamá-lo apenas uma vez após uma sequência de operações.

- **canvas:compose (x, y, src, [src_x, src_y, src_width, src_height]):** faz sobre o canvas (canvas de destino), em sua posição (x,y), a composição pixel a pixel com src (canvas de origem).

Onde:

x: [número] Posição da composição

y: [número] Posição da composição

src: [canvas] Canvas a ser composto

src_x: [número] Posição da composição no canvas src

src_y: [número] Posição da composição no canvas src

src_width: [número] Largura da composição no canvas src

`src_height`: [número] Altura da composição no canvas `src`

Os outros parâmetros entre colchetes são opcionais e indicam que parte do canvas `src` compor. Quando ausentes, o canvas inteiro é composto.

Essa operação chama `src:flush()` automaticamente antes da composição. E após a operação, o canvas de destino possui o resultado da composição e `src` não sofre qualquer alteração.

- **`canvas:pixel (x, y, R, G, B, A)`**: altera a cor de um pixel do canvas.

Onde:

`x`: [número] Posição do pixel

`y`: [número] Posição do pixel

`R`: [número] Componente vermelha da cor

`G`: [número] Componente verde da cor

`B`: [número] Componente azul da cor

`A`: [número] Componente alpha da cor

- **`canvas:pixel (x, y)`**: retorna a cor de um pixel do canvas.

Onde os parâmetros `x` e `y` funcionam como em `canvas:pixel`.

Retorna os valores `R`, `G`, `B` e `A`, nesta ordem.

- **`canvas:measureText (text)`**: retorna as coordenadas para o texto passado.

Onde:

`text`: [string] Texto a ser medido

Retorna `dx` e `dy`, largura e altura do texto respectivamente.

Prática

Vamos fazer um exemplo bem simples para melhor entendermos a utilização do módulo `event`.

O nosso exemplo irá mostrar na tela um botão, mas este só aparece por alguns instantes e depois some. Este botão irá aparecer quatro vezes e o objetivo é clicar nele. Se clicarmos nele pelo menos três vezes, será desenhado na tela um *retângulo verde* e a frase "You win", caso contrário será desenhado um *x vermelho* e a frase "You lose".

No nosso documento NCL, já criado no Capítulo 3, precisamos declarar nosso nó NCLua e este deve incluir duas propriedades conforme a figura 5.7.

```
<media id="clicks" src="clicks.lua">  
  <area id="counter"/>  
  <property name="inc"/>  
</media>
```

Figura 5.7: Nó NCLua com duas propriedades.

A âncora `counter`, ao ser inicializada, fará o código Lua contar quantas vezes o botão foi pressionado. A propriedade `inc` é um incrementador e será responsável por somar um ao valor de `counter`.

O elo para incrementar o contador é acionado na seleção do botão, que deve desaparecer (Figura 5.8).


```
<link xconnector="onSelectionStopSet">
  <bind role="onSelection" component="button"/>
  <bind role="stop" component="button"/>
  <bind role="set" component="clicks" interface="inc">
    <bindParam name="var" value="1"/>
  </bind>
</link>
```

Figura 5.8: Elo para incrementar o contador.

Conforme mostrado acima, o incremento é acionado por uma *simpleAction* de set. A ação de set é um apelido para uma ação de start em um evento do tipo attribution, portanto, o evento gerado é apresentado na Figura 5.9.

```
evt = {
  class    = 'ncl',
  type     = 'attribution',
  property = 'inc',
  action   = 'start',
  value    = '1',
}
```

Figura 5.9: Evento gerado pelo link.

Um novo conector foi criado (Figura 5.10).

```
<causalConnector id="onEndStopStart">
  <simpleCondition role="onEnd"/>
  <compoundAction operator="seq">
    <simpleAction role="stop" max="unbounded"/>
    <simpleAction role="start" max="unbounded"/>
  </compoundAction>
</causalConnector>
```

Figura 5.10: Novo conector criado.

Ele é responsável por fazer o elo entre o fim de *area04* da mídia *timere* o início de *counter* da mídia *clicks*.

Na Figura 5.11 podemos ver como ficou esse elo.

```
<link xconnector="onEndStopStart">
  <bind role="onEnd" component="timer" interface="area04"/>
  <bind role="stop" component="button"/>
  <bind role="start" component="clicks" interface="counter"/>
</link>
```

Figura 5.11: Novo elo.

Nosso código Lua pode ser visualizado na Figura 5.12.

```
counter = 0

local counterEvt = {
  class = 'ncl',
  type = 'attribution',
  name = 'counter',
}

function handler (evt)
  if((evt.class == 'ncl') and (evt.type == 'attribution')) then
    if (evt.name == 'inc') then
      counter = counter + evt.value
      event.post{
        class = 'ncl',
        type = 'attribution',
        name = 'inc',
        action = 'stop',
        value = counter,
      }
    end

    elseif(evt.label == 'counter') then
      if (counter >= 3) then
        canvas:attrFont('vera', 20)
        canvas:attrColor('green')
        canvas:drawText(280, 200, "You win")
        canvas:drawRect ('frame', 270, 230, 90, 70)
        canvas:flush()
      else
        canvas:attrFont('vera', 20)
        canvas:attrColor('red')
        canvas:drawText(280, 200, "You lose")
        canvas:drawLine(270, 230, 360, 300)
        canvas:drawLine(270, 300, 360, 230)
        canvas:flush()
      end
    end
  end

  event.register(handler)
```

Figura 5.12: Código Lua alterado.

5.3.3 Módulo settings

Propriedades de um nó settings só podem ser modificadas por meio de elos NCL. Não é permitido atribuir valores aos campos representando variáveis nos nós settings.

A tabela settings particiona seus grupos em várias subtabelas, correspondendo a cada grupo do nó application/x-ginga-settings. Por exemplo, em um objeto NCLua, a variável do nó settings "system.CPU" é referida como *settings.system.CPU*.

Exemplos de uso (Figura 5.13).

```
lang = settings.system.language
age = settings.user.age
val = settings.default.selBorderColor
settings.user.age = 18 --> ERRO!
```

Figura 5.13: Exemplo: módulo settings.

5.3.4 Módulo persistent

Aplicações NCLua permitem que dados sejam salvos em uma área restrita do middleware e recuperados entre execuções. O exibidor Lua define uma área reservada, inacessível a objetos NCL não procedurais. Não existe nenhuma variável pré-definida ou reservada e objetos procedurais podem atribuir valores a essas variáveis diretamente.

O uso da tabela persistent é semelhante ao uso da tabela settings, exceto pelo fato de que, neste caso, o código procedural pode mudar os valores dos campos.

Exemplos de uso (Figura 5.14).

```
persistent.service.total = 10
color = persistent.shared.color
```

Figura 5.14: Exemplo: módulo persistent.

Referências

- [1] F. Sant'Anna, R. Cerqueira, L.F.G. Soares. NCLua - Objetos Imperativos Lua na Linguagem Declarativa NCL.
- [2] ABNT NBR 15606-5. Televisão digital terrestre - Codificação de dados e especificações de transmissão para radiodifusão digital. Parte 5: Giga-NCL para receptores portáteis - Linguagem de aplicação XML para codificação de aplicações.

Apêndice A

Exemplos da Apostila


```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ncl id="exemplo01" xmlns="http://www.ncl.org.br/NCL3.0/EDTVProfile">

<head>

<regionBase>
  <region id="rg_video" left="0" top="0" width="100%"
        height="100%" zIndex="1">
    <region id="rg_button" left="40%" top="40%" width="20%"
        height="20%" zIndex="3"/>
  </region>
</regionBase>

<descriptorBase>
  <descriptor id="dp_video" region="rg_video"/>
  <descriptor id="dp_button" region="rg_button" focusIndex="0"/>
</descriptorBase>

<connectorBase>
  <causalConnector id="onBeginStart">
    <simpleCondition role="onBegin"/>
    <simpleAction role="start"/>
  </causalConnector>

  <causalConnector id="onEndStop">
    <simpleCondition role="onEnd"/>
    <simpleAction role="stop" max="unbounded"/>
  </causalConnector>

  <causalConnector id="onSelectionSet">
    <simpleCondition role="onSelection"/>
    <connectorParam name="var"/>
    <simpleAction role="set" value="$var"/>
  </causalConnector>
</connectorBase>

</head>
```

Figura A.1: Parte 1 do exemplo completo do Capítulo 3.

```
<body>
  <port component="video" id="inicio"/>

  <media id="video" src="video.mpg" type="video/mpeg" descriptor="dp_video">
    <area id="area01" begin="3s" end="6s"/>
    <area id="area02" begin="10s" end="13s"/>
    <area id="area03" begin="17s" end="20s"/>
    <area id="area04" begin="24s" end="27s"/>
  </media>

  <media id="button" descriptor="dp_button" src="button.jpg" type="image/jpeg"/>

  <media id="clicks" src="clicks.lua">
    <property name="inc"/>
    <property name="counter"/>
  </media>

  <link xconnector="onBeginStart">
    <bind role="onBegin" component="video"/>
    <bind role="start" component="clicks"/>
  </link>

  <link xconnector="onBeginStart">
    <bind role="onBegin" component="video" interface="area01"/>
    <bind role="start" component="button"/>
  </link>
  <link xconnector="onBeginStart">
    <bind role="onBegin" component="video" interface="area02"/>
    <bind role="start" component="button"/>
  </link>
  <link xconnector="onBeginStart">
    <bind role="onBegin" component="video" interface="area03"/>
    <bind role="start" component="button"/>
  </link>
  <link xconnector="onBeginStart">
    <bind role="onBegin" component="video" interface="area04"/>
    <bind role="start" component="button"/>
  </link>
```

Figura A.2: Parte 2 do exemplo completo do Capítulo 3.

```
<link xconnector="onEndStop">
  <bind role="onEnd" component="video" interface="area01"/>
  <bind role="stop" component="button"/>
</link>
<link xconnector="onEndStop">
  <bind role="onEnd" component="video" interface="area02"/>
  <bind role="stop" component="button"/>
</link>
<link xconnector="onEndStop">
  <bind role="onEnd" component="video" interface="area03"/>
  <bind role="stop" component="button"/>
</link>
<link xconnector="onEndStop">
  <bind role="onEnd" component="video" interface="area04"/>
  <bind role="stop" component="button"/>
</link>

<link xconnector="onSelectionStopSet">
  <bind role="onSelection" component="button"/>
  <bind role="set" component="clicks" interface="inc">
    <bindParam name="var" value="1"/>
  </bind>
</link>

</body>
</ncl>
```

Figura A.3: Parte 3 do exemplo completo do Capítulo 3.

```
--tabela
event = {}
event.post = {
  class = 'ncl',
  type = 'attribution',
  property = 'myProp',
  action = 'start',
  value = '10'
}

--funcao verificadora
function handler(evt)
  if evt.class ~= 'ncl' then return end
  if evt.type ~= 'attribution' then return end
  if evt.property ~= 'myProp' then return end

  print("Acao: ", evt.action)
  print("Valor: ", evt.value)

  return
end

--uso da funcao
handler(event) --> start    10
```

Figura A.4: Exemplo completo do Capítulo 4.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ncl id="exemplo01" xmlns="http://www.ncl.org.br/NCL3.0/EDTVProfile">
<head>

<regionBase>
  <region id="rg_video" left="0" top="0" width="100%"
        height="100%" zIndex="1">
    <region id="rg_button" left="40%" top="40%" width="20%"
        height="20%" zIndex="3"/>
  </region>
</regionBase>

<descriptorBase>
  <descriptor id="dp_video" region="rg_video"/>
  <descriptor id="dp_button" region="rg_button" focusIndex="0"/>
</descriptorBase>

<connectorBase>
  <causalConnector id="onBeginStart">
    <simpleCondition role="onBegin"/>
    <simpleAction role="start"/>
  </causalConnector>

  <causalConnector id="onEndStop">
    <simpleCondition role="onEnd"/>
    <simpleAction role="stop" max="unbounded"/>
  </causalConnector>

  <causalConnector id="onSelectionStopSet">
    <simpleCondition role="onSelection"/>
    <connectorParam name="var"/>
    <compoundAction operator="seq">
      <simpleAction role="stop" max="unbounded"/>
      <simpleAction role="set" value="$var"/>
    </compoundAction>
  </causalConnector>

  <causalConnector id="onEndStopStart">
    <simpleCondition role="onEnd"/>
    <compoundAction operator="seq">
      <simpleAction role="stop" max="unbounded"/>
      <simpleAction role="start" max="unbounded"/>
    </compoundAction>
  </causalConnector>
</connectorBase>

</head>
```

Figura A.5: Parte 1 do exemplo completo do Capítulo 5.


```
<body>
  <port component="video" id="inicio"/>

  <media id="video" src="video.mpg" type="video/mpeg" descriptor="dp_video">
    <area id="area01" begin="3s" end="6s"/>
    <area id="area02" begin="10s" end="13s"/>
    <area id="area03" begin="17s" end="20s"/>
    <area id="area04" begin="24s" end="27s"/>
  </media>

  <media id="button" descriptor="dp_button" src="button.jpg" type="image/jpeg"/>

  <media id="clicks" src="clicks.lua">
    <area id="counter"/>
    <property name="inc"/>
  </media>

  <link xconnector="onBeginStart">
    <bind role="onBegin" component="video"/>
    <bind role="start" component="clicks"/>
  </link>

  <link xconnector="onBeginStart">
    <bind role="onBegin" component="video" interface="area01"/>
    <bind role="start" component="button"/>
  </link>
  <link xconnector="onBeginStart">
    <bind role="onBegin" component="video" interface="area02"/>
    <bind role="start" component="button"/>
  </link>
  <link xconnector="onBeginStart">
    <bind role="onBegin" component="video" interface="area03"/>
    <bind role="start" component="button"/>
  </link>
  <link xconnector="onBeginStart">
    <bind role="onBegin" component="video" interface="area04"/>
    <bind role="start" component="button"/>
  </link>
```

Figura A.6: Parte 2 do exemplo completo do Capítulo 5.

```
<link xconnector="onEndStop">
  <bind role="onEnd" component="video" interface="area01"/>
  <bind role="stop" component="button"/>
</link>
<link xconnector="onEndStop">
  <bind role="onEnd" component="video" interface="area02"/>
  <bind role="stop" component="button"/>
</link>
<link xconnector="onEndStop">
  <bind role="onEnd" component="video" interface="area03"/>
  <bind role="stop" component="button"/>
</link>
<link xconnector="onEndStopStart">
  <bind role="onEnd" component="video" interface="area04"/>
  <bind role="stop" component="button"/>
  <bind role="start" component="clicks" interface="counter"/>
</link>

<link xconnector="onSelectionStopSet">
  <bind role="onSelection" component="button"/>
  <bind role="stop" component="button"/>
  <bind role="set" component="clicks" interface="inc">
    <bindParam name="var" value="1"/>
  </bind>
</link>

</body>
</ncl>
```

Figura A.7: Parte 3 do exemplo completo do Capítulo 5.

```
counter = 0

local counterEvt = {
  class = 'ncl',
  type = 'attribution',
  name = 'counter',
}

function handler (evt)
  if((evt.class == 'ncl') and (evt.type == 'attribution')) then
    if (evt.name == 'inc') then
      counter = counter + evt.value
      event.post{
        class = 'ncl',
        type = 'attribution',
        name = 'inc',
        action = 'stop',
        value = counter,
      }
    end

    elseif(evt.label == 'counter') then
      if (counter >= 3) then
        canvas:attrFont('vera', 20)
        canvas:attrColor('green')
        canvas:drawText(280, 200, "You win")
        canvas:drawRect ('frame', 270, 230, 90, 70)
        canvas:flush()
      else
        canvas:attrFont('vera', 20)
        canvas:attrColor('red')
        canvas:drawText(280, 200, "You lose")
        canvas:drawLine(270, 230, 360, 300)
        canvas:drawLine(270, 300, 360, 230)
        canvas:flush()
      end
    end
  end

  event.register(handler)
```

Figura A.8: Parte 4 do exemplo completo do Capítulo 5.

Apêndice B

Ferramentas

B.1 Composer

O Composer [1] é uma ferramenta de autoria hipermídia desenvolvida pelo Laboratório TeleMídia [2] do Departamento de Informática da PUC-Rio. Ele é um editor gráfico que fornece várias visões integradas para a criação de documentos NCL. As diversas visões facilitam a edição de documentos NCL por programadores com pouco conhecimento da linguagem NCL. A apresentação de um documento em diversas visões fornece ao usuário uma maior intuição no desenvolvimento de suas aplicações. A Figura B.1 apresenta o Composer e as diversas visões que o usuário pode trabalhar.

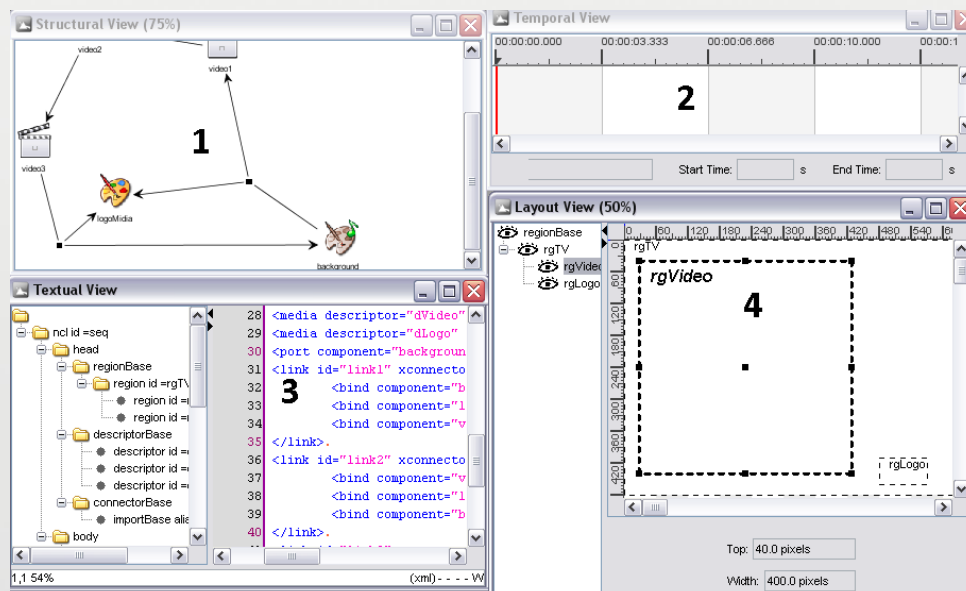


Figura B.1: Ferramenta de autoria Composer.

A versão atual do Composer permite ao usuário trabalhar com visões Estrutural 1, Temporal 2, Textual 3 e de Layout 4.

A Visão Estrutural (*Structural View*) apresenta os nós e os elos entre os nós. A Figura B.2 apresenta a visão estrutural de um documento NCL. Um nó de mídia é ilustrado por um ícone representando seu conteúdo. Uma composição é desenhada como uma caixa (cubo) e, quando ela é aberta (expanded), como um retângulo, com nós filhos e links dentro.

Nessa visão um autor pode graficamente criar, editar e remover nós de mídia, contextos e elos, bem como definir suas propriedades. Ele pode também expandir e fechar composições.

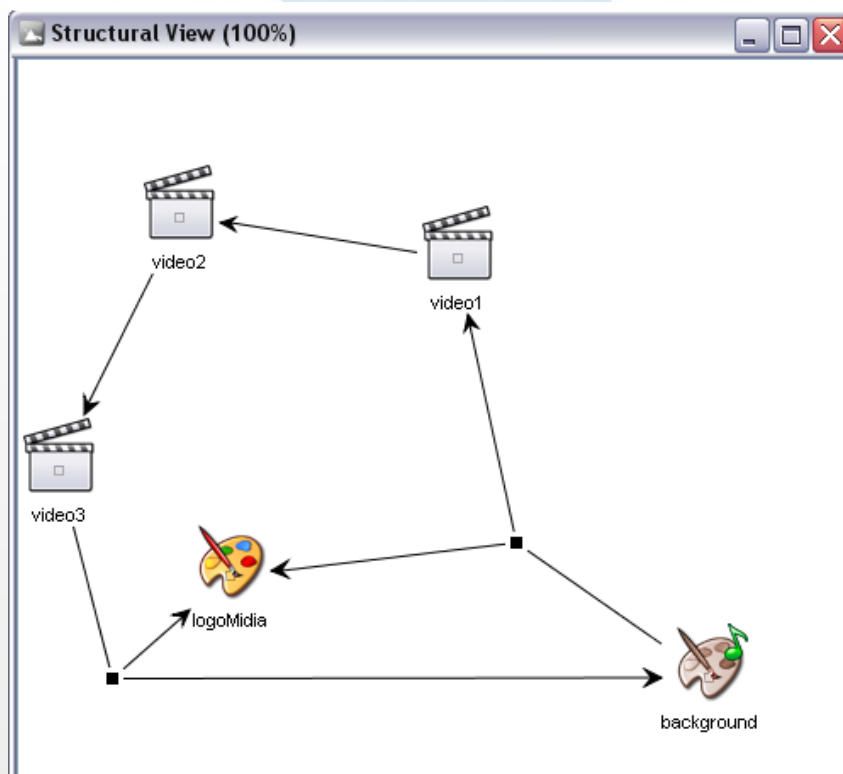


Figura B.2: Visão estrutural da ferramenta Composer.

A Visão Temporal (*Temporal View*) ilustra o sincronismo temporal entre os nós de mídia, e as oportunidades de interatividade. A visão temporal preserva tanto quanto possível o paradigma de linha de tempo. Usando essa visão, a autoria pode ser feita colocando objetos de mídia sobre um eixo de tempo, mas preservando os relacionamentos relativos entre eles. A Figura B.3 apresenta a visão temporal de um documento hipermídia que exibe uma imagem *iconeInteracao* durante um determinado segmento de apresentação da mídia *videoPrinc*.

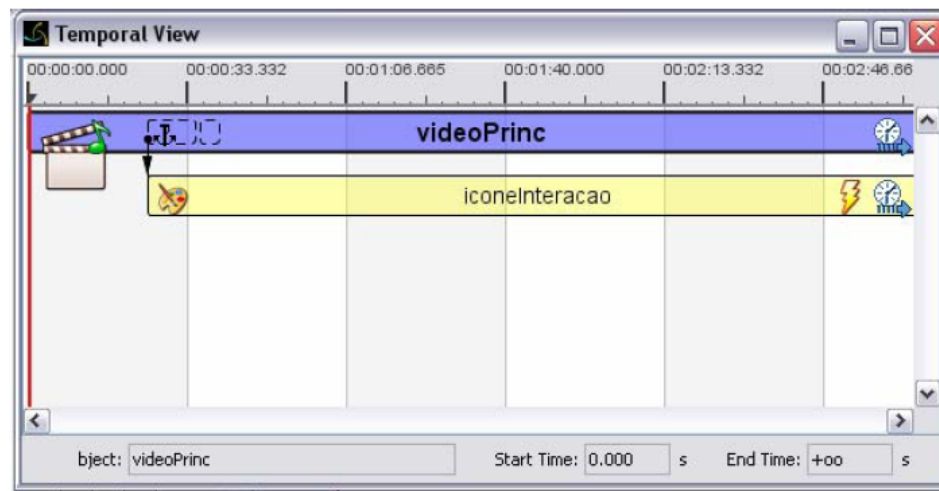


Figura B.3: Visão temporal da ferramenta Composer.

A Visão de Leiaute (*Layout View*) exibe as regiões da tela onde as mídias do documento poderão ser apresentadas. Ela é dividida em duas partes. Considerando que uma região pode ser especificada como filha de outra região, do lado esquerdo da visão de leiaute é mostrado a organização hierárquica de regiões. Do lado direito é exibido a representação espacial. Quando a região pai é movida, todas as regiões filhas também são movidas, preservando suas posições relativas. A Figura B.4 apresenta a visão de leiaute de um documento hipermídia que contém três regiões: a tela inteira (*rgTV*), uma área para exibir vídeos (*rgVideo*) e uma área que exibe um logo (*rgLogo*).

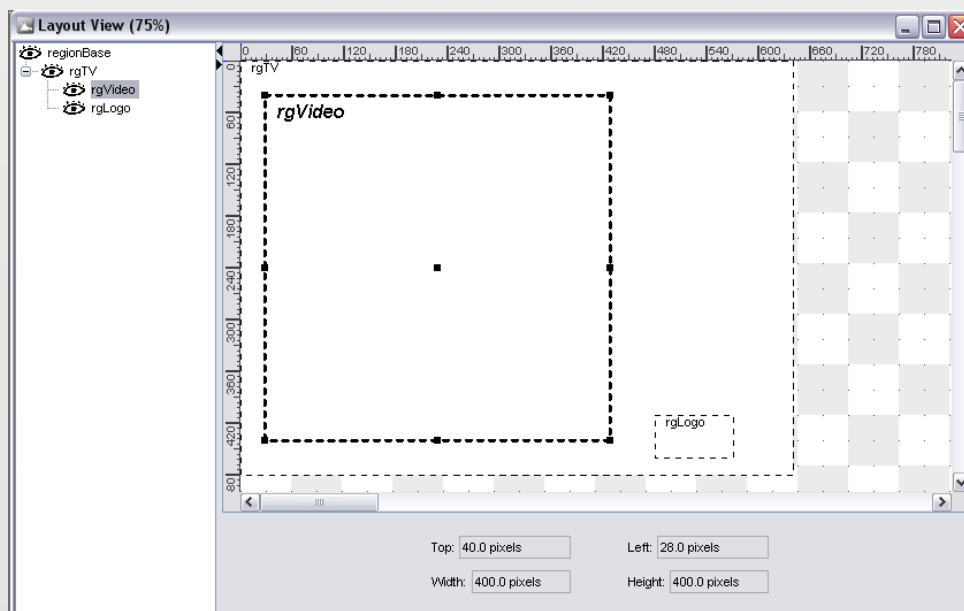


Figura B.4: Visão de leiaute da ferramenta Composer.

A Visão Textual (*Textual View*) faz a apresentação textual do código NCL. Nessa visão, o usuário pode editar diretamente o código NCL como em um editor de texto. Ela oferece algumas funcionalidades para a edição de documentos NCL, como visão em árvore do código, coloração

de palavras reservadas da linguagem. A figura B.5 apresenta a visão textual.

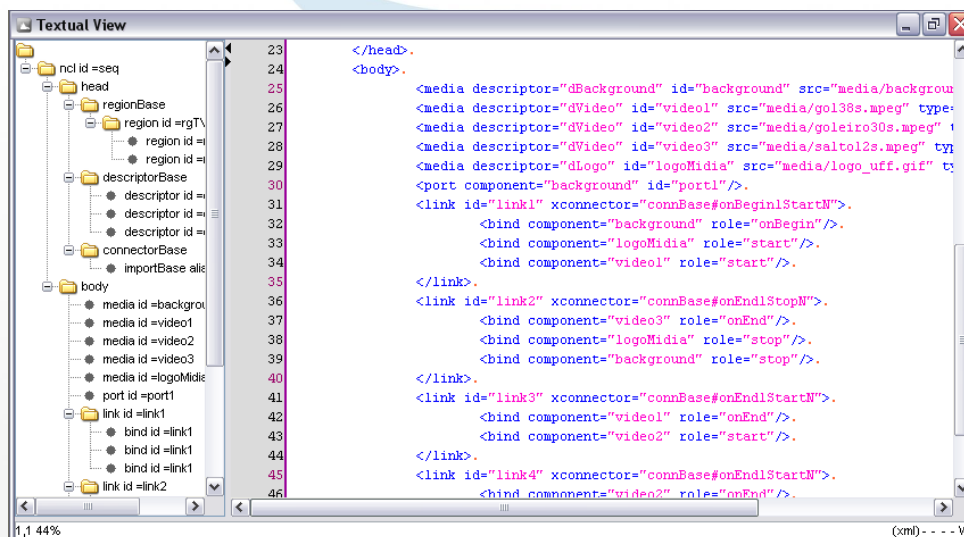


Figura B.5: Visão textual da ferramenta Composer.

As visões do Composer são integradas, pois sempre que são feitas alterações em uma visão, elas são refletidas nas demais visões da ferramenta.

B.2 NCL Eclipse

NCL Eclipse é um editor XML/NCL desenvolvido como um plug-in para a plataforma Eclipse [3] que oferece diversas funcionalidades centradas nas necessidades de autores de aplicações NCL, tais como a sugestão de código automática e contextual, validação de documentos NCL, coloração de elementos XML e de palavras reservadas da linguagem.

Ao ser desenvolvido como um plug-in para o Eclipse o NCL Eclipse permite que todo esse ambiente, já bem conhecido dos programadores, seja reutilizado e facilita a integração com outras ferramentas de desenvolvimento para a TV Digital, como, por exemplo, o Lua Eclipse [4].

Uma das funcionalidades interessantes no desenvolvimento de programas com o NCL Eclipse é a identificação e marcação de erros no momento da autoria, permitindo que o autor identifique o erro de forma rápida, visto que diminui a necessidade de executar o documento para evidenciar que o mesmo possui erros sintáticos e/ou semânticos. O NCL Eclipse reusa o NCL Validator com o objetivo de identificar e marcar erros em documentos NCL, em tempo de autoria.

O processo de instalação do NCL Eclipse é muito simples e segue o padrão da instalação de plug-ins para o Eclipse. No site <http://laws.deinf.ufma.br/ncl eclipse/documentacao.htm> existe um tutorial básico sobre como instalar e usar o NCL Eclipse.

A Figura B.6 apresenta uma visão geral do NCL Eclipse com um documento NCL recém criado.

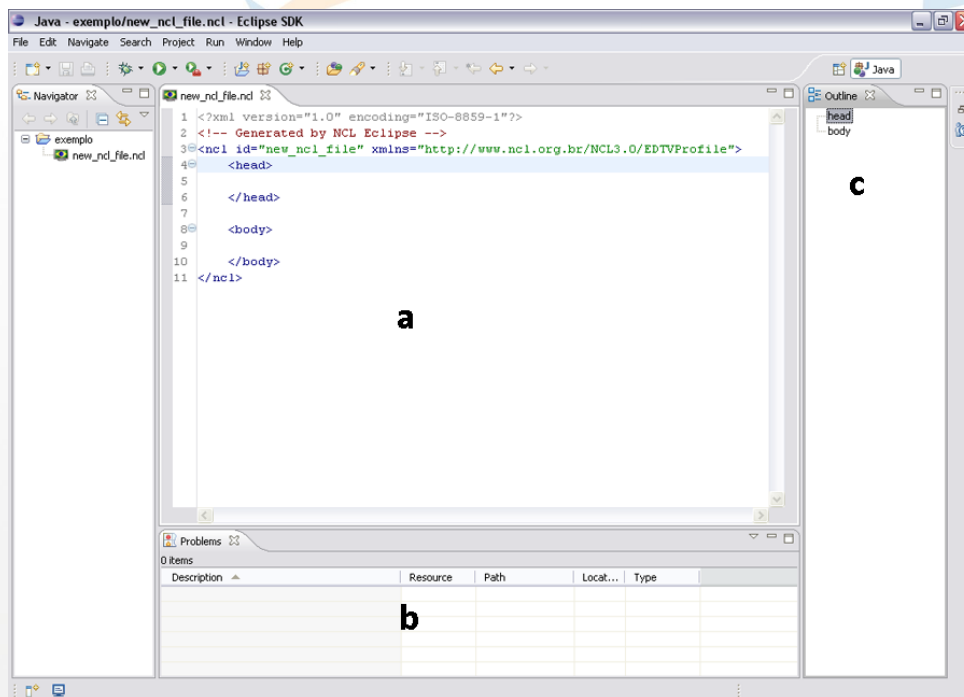


Figura B.6: Visão geral do NCL Eclipse: a) visão textual; b) visão de problemas; c) visão outline.

A Figura B.7 apresenta um documento com um erro, a linha do erro é sublinhada e uma descrição pode ser visualizada na *Problem View*.

Na autoria de documentos o autor pode utilizar a opção de autocomplete. O programador precisa apenas colocar o cursor em um trecho de texto interno ao conteúdo de um determinado elemento e teclar “Ctrl + espaço” para visualizar uma sugestão dos possíveis elementos filhos. A Figura B.8 apresenta a opção de autocomplete.

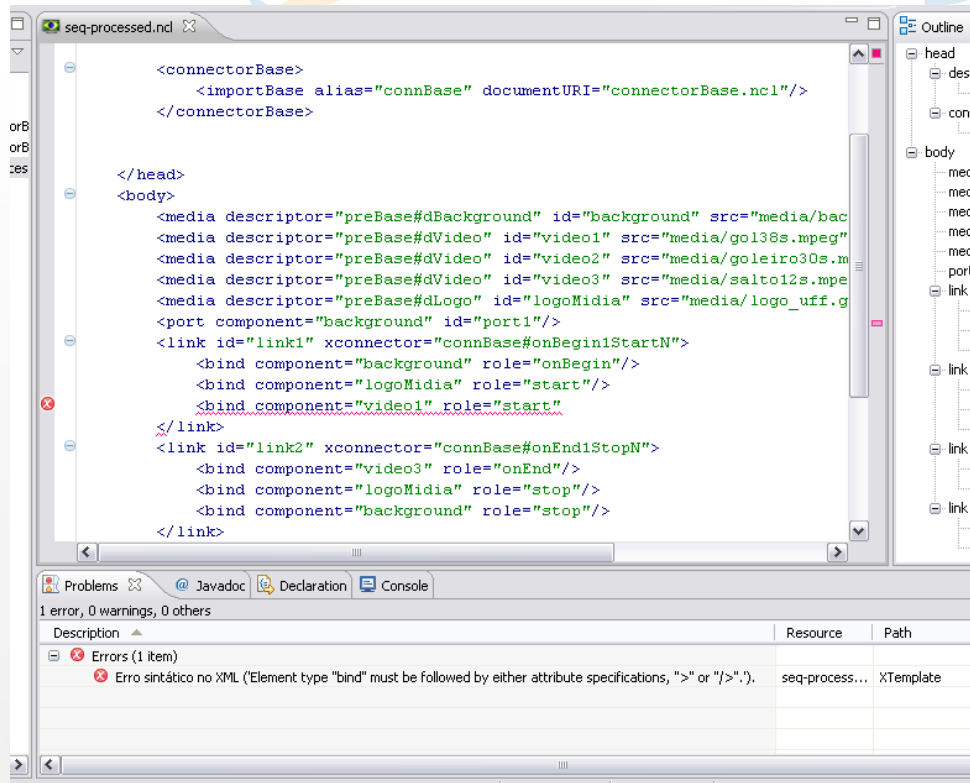


Figura B.7: Identificação e marcação de erros em documentos NCL.

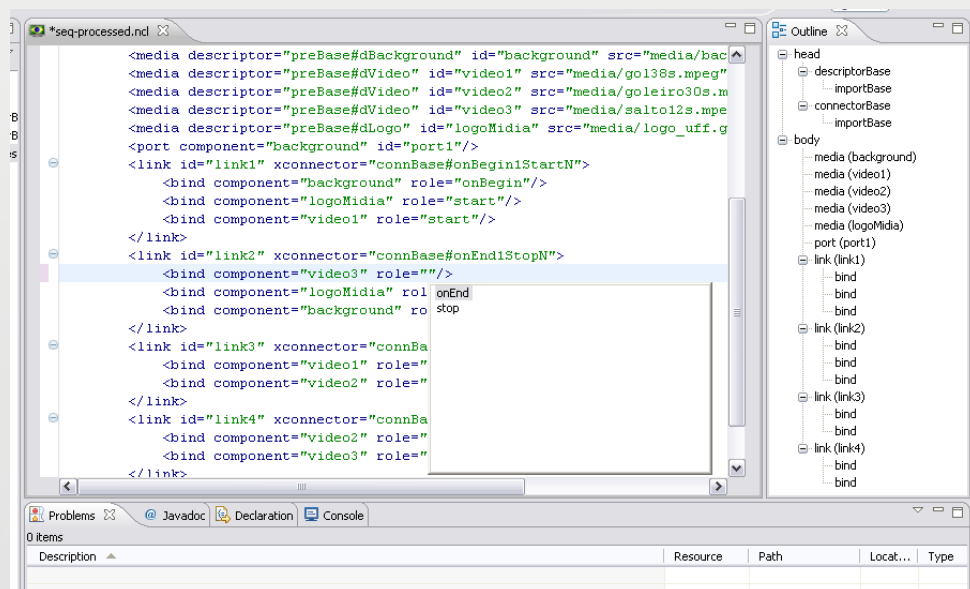


Figura B.8: Sugestão de código contextual automática (Autocomplete).

Referências

- [1] Construindo Programas Audiovisuais Interativos Utilizando a NCL 3.0 e a Ferramenta Composer - 2a. edição (versão 3.0), PUC-Rio, 31/07/2007.



- [2] TeleMídia (<http://www.telemidia.puc-rio.br>). Acessado em 14 de outubro de 2009.
- [3] Eclipse (<http://www.eclipse.org>). Acessado em 19 de outubro de 2009.
- [4] LuaEclipse (<http://luaeclipse.luaforge.net>). Acessado em 19 de outubro de 2009.