# 3
# A Linguagem LindaX

O principal objetivo deste capítulo é apresentar a estrutura e principais características da linguagem de especificação de arquiteturas LindaX. É nessa linguagem que está centrada a proposta da tese presente. Para atender os requisitos de um ambiente de criação de serviços mencionados no Capítulo 2, a linguagem se apóia fundamentalmente no conceito de 'estilos arquiteturais' (Monroe et al., 1997). A adoção dessa estratégia surgiu da percepção de que, em relação aos requisitos mencionados, estilos possibilitam: (i) a definição de bibliotecas de tipos de entidades básicas que possam ser reusados em diferentes projetos, (ii) a prescrição, com níveis seletivos de formalismo, de características comportamentais e arquiteturais de sistemas, em termos de suas entidades constituintes e das restrições de composição entre essas entidades e (iii) a concepção de ferramentas que permitam automatizar a configuração de sistemas e a implementação de mecanismos de controle de adaptações nesses sistemas.

A partir da descrição de estilos em LindaX, pode-se estabelecer DSLs diferenciadas semanticamente, que tratam de aspectos específicos de um sistema de comunicação. Essas DSLs são especializações de um mesmo arcabouço sintático para descrições arquiteturais provido pela linguagem LindaX. Ferramentas extensoras (vide exemplos no Capítulo 4) são responsáveis por dar a uma DSL particular a semântica embutida no estilo a ela associado.

A estrutura sintática de estilos e DSLs LindaX é inteiramente baseada em esquemas XML (World Wide Web Consortium, 2001), ou seja, descrições em LindaX são de fato documentos XML. Quatro esquemas-base XML constituem o núcleo da linguagem:

1. `lindaxtyp`: define um sistema de tipos para a linguagem. Tipos LindaX permitem a prescrição de características comuns (por exemplo, comportamento esperado, pontos de adaptação, implementação etc.) a entidades de computação e comunicação presentes na arquitetura de um

sistema (descrita a partir do esquema `lindaxcnf`). Essas entidades personificam as abstrações da visão arquitetural do modelo SCM, apresentado no Capítulo 2;

2. `lindaxcnf`: provê artefatos que permitem descrever a arquitetura de sistemas em termos de composições entre entidades cujos tipos são definidos a partir do esquema `lindaxtyp`. Esses artefatos constituem o arcabouço sintático para as DSLs LindaX;
3. `lindaxres`: permite a especificação de uma visão de recursos associada a uma arquitetura descrita a partir do esquema `lindaxcnf`;
4. `lindaxsty`: oferece uma estrutura de especificação de estilos LindaX. O vocabulário de entidades de um estilo é composto por tipos descritos a partir do esquema `lindaxtyp`. As restrições de um estilo são predicados que determinam a validade ou não das descrições de arquitetura associadas ao estilo.



Todos esses esquemas oferecem estruturas genéricas desprovidas (a princípio) de informações acerca do comportamento das entidades de computação e comunicação ou de aspectos de implementação, entre outros. No entanto, esses esquemas são dotados de estruturas para propriedades arbitrárias – definidas em um esquema "utilitário", chamado `lindaxprop`, o qual todos os esquemas-base importam – cujos valores podem ser processados por ferramentas de cunho específico, como tradutores para determinadas notações ou geradores automáticos de código para diferentes plataformas de programação.

O restante deste capítulo apresenta em detalhe os esquemas-base de LindaX. Para tornar mais clara a apresentação dos esquemas, uma notação sem as *tags* e *namespaces* normalmente empregados em documentos XML é utilizada nos exemplos de especificação ilustrados ao longo do capítulo. Por comodidade aos usuários da linguagem, editores dessa notação são oferecidos na implementação do ambiente LindaStudio (vide Capítulo 5). A definição completa dos esquemas apresentados ao longo deste capítulo está disponível no Apêndice C.

## 3.1. Estrutura de propriedades (esquema utilitário `lindaxprop`)

Diversos elementos XML definidos nos esquemas-base de LindaX permitem a declaração de propriedades arbitrárias. O esquema `lindaxprop`, que é importado pelos outros esquemas-base, define o formato das propriedades como um tipo complexo XML contendo uma dupla `{nome,valor}`. Em um documento XML, uma propriedade é declarada como na Figura 3.1.

```
1  ...
2   <property name="nome"
3             xsi:type="lindaxprop:Property">
4     <value xsi:type="lindaxprop:...">
5       <!-- elemento XML descrevendo valor -->
6     </value>
7   </property>
8  ...
```

Figura 3.1. Exemplo de declaração em XML de uma propriedade LindaX.

Essa mesma dupla é representada na notação sem *tags* por

```
nome = valor;
```

O atributo XML `nome`, usado para identificar a propriedade, pode ser qualquer cadeia de caracteres aceita pelo tipo nativo `string` de XML. O elemento XML `valor` pode ser de um dos cinco tipos indicados na Tabela 3.1.

Tabela 3.1. Tipos de valores para propriedades em LindaX

| Tipo | Valor | Notação sem *tags* | Fonte |
|---|---|---|---|
| `string` | Cadeia de caracteres | `"uma string"` | Tipo nativo XML |
| `hexBinary` | Número com quantidade arbitrária de bits | `1` | Tipo nativo XML |
| `List` | Lista de elementos de qualquer um dos outros tipos | `{"string a","string b"}`<br>`{1,2}`<br>`{"string a",1}`<br>`{#id_elem_1,#id_elem2}` | Tipo de `lindaxprop` |
| `Range` | Intervalo de números | `{5..10}` (fechado)<br>`{5..}` (aberto à dir.)<br>`{5..#val}` {parametrizado} | Tipo de `lindaxprop` |
| `ExternalRef` | Referência ao identificador XML de outro elemento | `#id_do_elemento` | Tipo de `lindaxprop` |

O tipo `ExternalRef` merece atenção especial. Ele é definido como um tipo complexo XML contendo a tupla `{externalSymbol,param1,...,paramN}`. O elemento `externalSymbol` (tipo nativo XML `anyURI`) permite referenciar outros elementos XML por meio de seus identificadores (atributos XML `id` do tipo

nativo XML `ID`) Os outros elementos de `ExternalRef` definem uma lista (possivelmente vazia) de outros elementos de um dos tipos da Tabela 3.1. Essa lista serve como um conjunto de parâmetros que podem ser passados ao elemento referenciado por `externalSymbol`. Isso é particularmente útil na descrição de entidades computacionais e de comunicação das quais algumas características não se quer definir diretamente na descrição– por exemplo, a identificação do tipo de componente de software ou da classe de objeto[1] que implementa a entidade. Isso propicia um melhor reuso da mesma.

## 3.2. Sistema de tipos (esquema-base `lindaxtyp`)

As abstrações de componente e *pipe* no modelo SCM (Capítulo 2) são essencialmente estruturais e semanticamente neutras. Nenhuma forma de descrição de comportamento dessas abstrações é provida no modelo. Desse modo, à luz da engenharia de software, componentes e *pipes* seriam representações cujas manifestações pode-se avaliar somente em tempo de execução. O esquema `lindaxtyp` define um sistema de tipos a partir do qual informações acerca das abstrações arquiteturais do modelo podem ser especificadas em tempo de projeto.

O sistema de tipos de LindaX consiste em tipos de componente, *pipe* e interface, que são definidos no esquema `lindaxtyp` pelos tipos complexos XML `ComponentType`, `PipeType` e `InterfaceType`. Na versão atual da linguagem, somente tipos para componentes e *pipes* primitivos podem ser declarados. Porém, composições arquiteturais podem ser prescritas em LindaX por intermédio de *templates* (vistos na Seção 3.3).

Cada tipo de componente possui um conjunto de descritores de portas que indicam o que as instâncias desse tipo esperam de (ou oferecem a) outros componentes. De modo similar, cada tipo de *pipe* possui um conjunto de descritores de pontos de acesso (*pipes* podem ser multiponto*)* que indicam os papéis esperados dos componentes ligados a instâncias desse tipo. Nem todas as portas e pontos de acesso "desejáveis" precisam ser necessariamente descritos em um determinado tipo de componente ou *pipe*, tampouco uma instância de componente ou *pipe* precisa implementar somente as portas e pontos de acesso

[1] Há na literatura atual sobre componentes de software uma imensa confusão acerca do uso dos termos 'tipo de componente', 'componente' e 'instância de componente'. Nesta tese, os termos 'tipo de componente' e 'componente' são usados para designar, respectivamente, tipo e instância.

descritos em seus tipos. Descrições de portas e pontos de acesso identificam somente o 'mínimo' de pontos de interação que devem ser providos pelas instâncias do tipo de componente ou *pipe* correspondente. Restrições semelhantes, porém mais flexíveis, na configuração de pontos de interação de componentes e *pipes* podem ser estabelecidas a partir de estilos (vide Seção 3.5).

Cada descritor de porta ou ponto de acesso (tipos complexos XML `Port` e `AccessPoint`) tem um conjunto de assinaturas (tipo complexo XML `Signature`) que o associam a determinados tipos de interface, por meio da propriedade `Type` (tipo XML `ExternalRef`).[2] Um tipo de interface prescreve um padrão específico de interação que deve ser respeitado por uma porta ou ponto de acesso.

Todo tipo de componente, *pipe* e interface possui um identificador XML unívoco, assim como todo descritor de porta, ponto de acesso e assinatura. Todos esses elementos podem ser adornados com propriedades adicionais que descrevem suas características. Por exemplo, as propriedades `Cardinality` (tipo XML `hexBinary` ou `Range`) e `Level` (tipo XML `string`) são usadas, respectivamente, para limitar o número de instâncias criadas a partir de um tipo (ou o número de pontos de interação criados a partir de uma descrição de porta ou ponto de acesso) e para identificar portas de nível meta.[3] A princípio, o sistema de tipos de LindaX é semanticamente neutro, não restringindo os padrões de computação e de interação que podem ser descritos. Novas propriedades e ferramentas que lidam com essas propriedades podem ser definidas, com vistas a fornecer maiores informações sobre os tipos. Por exemplo, a propriedade `Direction` (tipo XML `string`), atribuída a assinaturas, é utilizada por uma das ferramentas de síntese do ambiente LindaStudio para mapear tipos de interfaces em LindaX para receptáculos na plataforma de componentes OpenCOM (Coulson et al., 2004).[4] Detalhes desse mapeamento são apresentados no Capítulo 5.

Tipos de componente, *pipe* e interface podem ser parametrizados. Em termos da notação XML, os tipos complexos XML correspondentes a essas entidades possuem elementos do tipo simples XML `Parameter`, que estende o tipo nativo XML `ID`. Esses parâmetros são geralmente referenciados em valores de propriedades do tipo XML `ExternalRef`. Isso permite que uma mesma propriedade seja usada na integração de LindaX com diferentes formalismos ou

[2] Quando uma porta apresenta uma única assinatura, as propriedades dessa assinatura podem ser atribuídas diretamente à porta. O mesmo se aplica a pontos de acesso.

[3] Na ausência dessas propriedades, assume-se `Cardinality = {0..}` (para portas e pontos de acesso, `Cardinality = {1..}`) e `Level = "base"`.

[4] Na ausência dessa propriedade, assume-se `Direction = "any"`.

técnicas de síntese. Por exemplo, a propriedade `Behaviour` pode ser usada por diferentes ferramentas de refinamento e síntese. Uma das ferramentas de síntese implementadas no ambiente LindaStudio usa a propriedade `Behaviour` na associação de tipos LindaX a componentes e interfaces da plataforma OpenCOM, que são identificáveis por números inteiros de 128 bits. Essa mesma propriedade poderia ser usada, porém, na associação com classes e interfaces Java, que são identificáveis por cadeias de caracteres. O fato dos parâmetros serem representados, em última instância, como identificadores XML força que os valores associados a esses parâmetros não sejam definidos na própria especificação de um tipo, guiando o projetista em um melhor reuso dessa especificação. Esses valores são geralmente descritos em arquivos de propriedades específicos que alimentam cada ferramenta distinta. Esse processo é explicado mais detalhadamente no Capítulo 5.

A gramática em EBNF (Int'l Organisation for Standardisation, 1996) da notação sem *tags* do sistema de tipos de LindaX é apresentada na Figura 3.2. Para fins comparativos, a Figura 3.11 esboça a definição de um tipo de interface em LindaX usando tanto XML quanto a notação sem *tags*.[5]

```
1  TypeDecl      := CompTypeDecl | PipeTypeDecl | IntfTypeDecl
2  CompTypeDecl  := "ComponentType" Symbol ( "(" ParamList ")" )?
                    "{" CompTypeBody "}"
3  PipeTypeDecl  := "PipeType" Symbol ( "(" ParamList ")" )?
                    "{" PipeTypeBody "}"
4  IntfTypeDecl  := "InterfaceType" Symbol ( "(" ParameterList ")" )?
                    "{" IntfTypeBody "}"
5  ParamList     := Symbol ( "," Symbol )*
6  CompTypeBody  := PropertyDecls PortDecls
7  PipeTypeBody  := PropertyDecls APDecls
8  IntfTypeBody  := PropertyDecls
9  PortDecls     := ( PortDecl )*
10 PortDecl      := "Port" Symbol "{" PropertyDecls SignDecls "}"
11 APDecls       := ( APDecl )*
12 APDecl        := "AccessPoint" Symbol "{" PropertyDecls SignDecls "}"
13 SignDecls     := ( SignDecl )*
14 SignDecl      := "Signature" Symbol "{" PropertyDecls "}"
15 PropertyDecls := ( PropertyDecl )*
16 PropertyDecl  := Symbol "=" ElemExp ";"
17 ElemExp       := StrVal | IntVal | SetExp | ExternalParam
18 SetExp        := "{" ( RangeExp | ElemList )? "}"
19 RangeExp      := IntVal ".." ( IntVal )?
20 ElemList      := ElementExp ( "," ElementExp )*
21 ExternalParam := "#" <IDENTIFIER> ( "." <IDENTIFIER> )*
22 Symbol        := <IDENTIFIER>
23 StrVal        := <STRING_LITERAL>
24 IntVal        := <INTEGER_LITERAL>
```

Figura 3.2. Gramática da notação sem *tags* do sistema de tipos de LindaX.

[5] Na verdade, a sintaxe XML para definição de tipos de componente, *pipe* e interface é bem mais complexa que a apresentada no exemplo. Privilegiou-se, no entanto, uma visão geral da estrutura em detrimento da apresentação de níveis intermediários de aninhamento de elementos XML. Esses detalhes são apresentados no Apêndice C.

```
   ===> NOTAÇÃO XML
1  <interfaceType id="PPacketPush"
2                 xsi:type="lindaxtyp:InterfaceType">
3    <property name="Behaviour"
4              xsi:type="lindaxprop:Property">
5       <value xsi:type="lindaxprop:ExternalRef">
6        <externalSymbol href="#PPacketPush.behv"
7                        xsi:type="xsd:anyURI"/>
8      </value>
9    </property>
10   <parameter id="behv"
11             xsi:type="lindaxtyp:Parameter"/>
12 </interfaceType>
    - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   ===> NOTAÇÃO SEM TAGS
1  InterfaceType PPacketPush( behv ) { Behaviour = #behv; }
```

Figura 3.3. Exemplo de descrição em XML e na notação sem *tags* em LindaX.

A Figura 3.4 descreve, na notação sem *tags,* os tipos de componente, *pipe* e interface usados na representação do módulo IP de um roteador implementado por software. Uma representação diagramática das entidades que constituem esse módulo e seus tipos correspondentes é exemplificada na Figura 3.5.

```
1  InterfaceType PPacketPush( behv ) { Behaviour = #behv; }
2  InterfaceType PPacketPull( behv ) { Behaviour = #behv; }
3  InterfaceType IMetaFilter( behv ) { Behaviour = #behv; }
4
5  ComponentType UIOModule( behv ) {
6    Behaviour = #behv;
7    Port io {
8      Cardinality = 2;
9      Signature in  { Type = #PPacketPush; Direction = "in"; }
10     Signature out { Type = #PPacketPull; Direction = "in"; }
11   }
12   Port filter_ml {
13     Level = "meta"; Cardinality = 1;
14     Signature flt { Type = #IMetaFilter; Direction = "in"; }
15   }
16 }
17
18 ComponentType UForwardingModule( behv,nports ) {
19   Behaviour = #behv;
20   Port io {
21     Cardinality = #nports;
22     Signature in  { Type = #PPacketPull; Direction = "out"; }
23     Signature out { Type = #PPacketPush; Direction = "out"; }
24   }
25 }
26
27 PipeType SLocalEnv( behv ) {
28   Behaviour = #behv;
29   AccessPoint io {
30     Cardinality = 1;
31     Signature in  { Type = #PPacketPull; Direction = "out"; }
32     Signature out { Type = #PPacketPush; Direction = "out"; }
33   }
34   AccessPoint fwd {
35     Cardinality = 1;
36     Signature in  { Type = #PPacketPush; Direction = "in"; }
37     Signature out { Type = #PPacketPull; Direction = "in"; }
38   }
39 }
```

Figura 3.4. Definição de tipos de entidades de um módulo IP em LindaX.

Na Figura 3.4 são declarados, primeiramente, tipos de interface de passagem de pacotes – para envio (*push*, linha 1) ou solicitação (*pull*, linha 2) – e de configuração de filtros de pacotes (linha 3). A seguir, há a declaração de um tipo para módulos de entrada e saída (E/S) das redes físicas subjacentes (linhas 5 a 16). Cada módulo de E/S implementa exatamente duas portas de passagem de pacotes e uma porta de nível meta que permite a configuração de filtros de pacotes no mesmo. É declarado também um tipo para módulos de encaminhamento (linhas 18 a 25). Um módulo de encaminhamento é dotado de um número variável de portas para passagem de pacotes (esse número é determinado pelo parâmetro `nports`). Finalmente, é declarado um tipo para o *pipe* interligando esses componentes (linhas 27 a 39) com exatamente dois pontos de acesso para passagem de pacotes. Como pode ser observado, módulos de E/S são "passivos" no exemplo (pacotes são entregues e solicitados a eles), enquanto módulos de encaminhamento são "ativos" (eles solicitam pacotes aos módulos de entrada e os entregam aos módulos de saída).



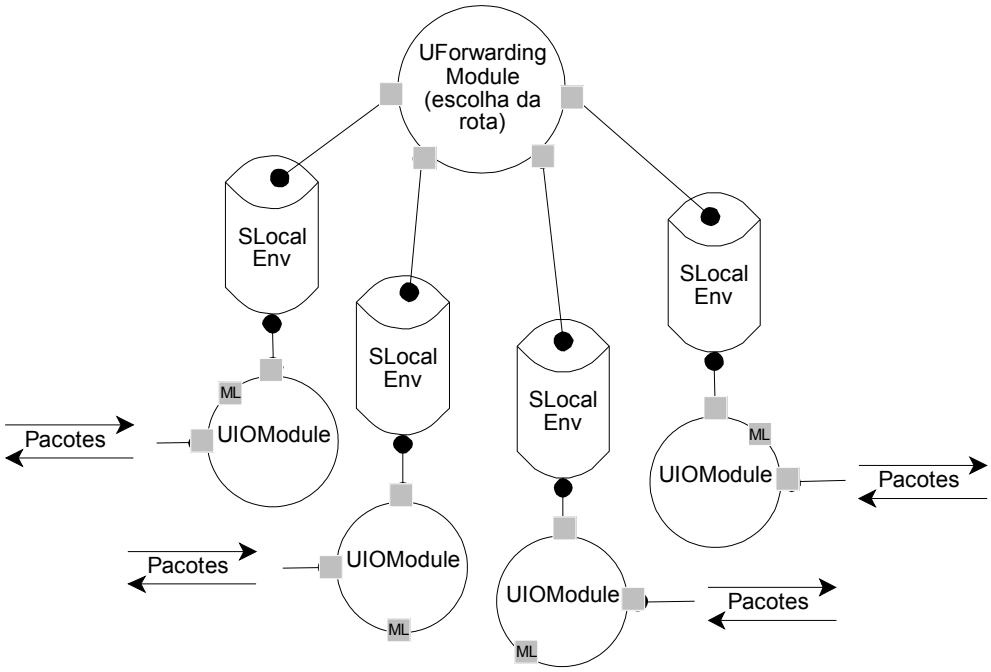


Figura 3.5. Diagrama SCM do módulo IP de um roteador.

## 3.3. Descrições de arquitetura (esquema base `lindaxcnf`)

O esquema `lindaxcnf` permite a descrição de arquiteturas a partir do sistema de tipos definido pelo esquema `lindaxtyp`. Descrições de arquitetura em LindaX incluem sistemas e *templates*, que são definidos no esquema pelos tipos complexos XML `System` e `SystemTemplate`. Sistemas e *templates* definem configurações de software. Um sistema descreve uma configuração específica, ao passo que um *template* prescreve um "molde" a partir do qual várias configurações similares podem ser instanciadas. Todo sistema e *template* possui um identificador XML unívoco.

Cada sistema possui um conjunto de descritores de instâncias (dotados também de identificadores XML) que indicam os constituintes da configuração que ele descreve. O mesmo se aplica a *templates* e às configurações que eles prescrevem. Cada descritor de instância (definido pelo tipo complexo XML `Instance`) contém uma lista não vazia de propriedades. A única propriedade obrigatória nessa lista é `Type` (tipo XML `ExternalRef`)[6], que faz referência a um dos seguintes elementos: (i) um tipo de componente declarado em uma especificação de tipos, ou (ii) um *template*, indicando que uma instância de um molde de configuração é aninhada em outra configuração mais externa.

Um sistema pode ser aninhado em uma configuração mais externa, por meio de um descritor de inclusão (tipo complexo XML `Include`). Cada descritor de inclusão contém uma lista não vazia de propriedades, das quais a única obrigatória é `Type` (tipo XML `ExternalRef`), que referencia o sistema aninhado.

É importante notar que o esquema `lindaxcnf` não restringe a descrição de configurações que compartilham constituintes em comum. Ou seja, descritores de inclusão em configurações distintas podem referenciar um mesmo sistema. Quando um *template* inclui um sistema, todas as instâncias desse *template* compartilham esse mesmo sistema. Obviamente, a adequação ou não de configurações com compartilhamento de constituintes depende do alvo das descrições de arquitetura em LindaX. Por exemplo, poucos modelos de componentes, como Fractal (Bruneton et al., 2002) e Enrie

[6] Por causa da forma genérica como propriedades são definidas em LindaX (esquema `lindaxprop`), a obrigatoriedade dessa propriedade – e de outras apresentadas ao longo do capítulo – só é conseguida a partir das ferramentas que manipulam as descrições de arquitetura em LindaX, não pela conformidade dessas descrições em relação ao esquema `lindaxcnf`.

(Outhred e Potter, 1998), permitem nativamente essa forma de configuração. O mesmo pode ser dito a respeito das ADLs. Durante os processos de refinamento e síntese de código as ferramentas associadas podem planificar essas configurações se necessário, removendo os aninhamentos (vide Capítulo 5).

Sistemas e *templates* também possuem conjuntos de descritores de *pipes* (definidos pelo tipo complexo XML `Pipe`) que permitem estabelecer associações entre seus componentes e configurações constituintes. O identificador XML de um *pipe* é opcional, ou seja, *pipes* podem ser anônimos. Cada descritor de *pipe* contém uma lista de elementos do tipo XML `ExternalRef`, que identifica os componentes e configurações associados ao *pipe*. Esse descritor também possui uma lista de propriedades, que pode ser vazia. Uma vez que *pipes* são declarados internamente em sistemas e *templates*, é sempre necessária a descrição de um sistema mais externo que engloba todos os elementos de uma configuração.

Por fim, sistemas e *templates* possuem conjuntos de descritores de mapeamentos (tipo complexo XML `Mapping`). Mapeamentos indicam que alguns constituintes de um sistema ou *template* são acessíveis externamente a essa configuração – por *default*, constituintes de uma configuração são inacessíveis em LindaX. Assim como para sistemas, *templates* e descritores de instâncias, descritores de mapeamentos possuem obrigatoriamente um identificador XML.

Descrições de sistemas e *templates* também podem ser adornadas com propriedades que descrevem suas características. A única propriedade obrigatória nessas descrições é `Style` (tipo XML `ExternalRef`), que faz referência ao estilo arquitetural a ser seguido pela configuração descrita (vide Seção 3.5). Para manter a linguagem a mais simples possível, não é permitido atribuir informações adicionais acerca de comportamentos e aspectos de implementação, além daquelas definidas pelos componentes constituintes. Adicionalmente, sistemas e *templates* não definem pontos de interação próprios, além daqueles de constituintes que tenham sido exteriorizados por meio de mapeamentos. Em última instância, essa restrição implica em que pontos de interação entre sistemas e *templates* sejam sempre representados a partir de mapeamentos (possivelmente sucessivos) de portas dos componentes mais internos da configuração. Apesar de limitar a expressividade da linguagem, essas restrições simplificam consideravelmente refinamentos e sínteses de descrições de arquiteturas para ADLs e plataformas de programação – principalmente quando planificações são necessárias.

A gramática em EBNF da notação sem *tags* para descrições de arquitetura em LindaX é apresentada na Figura 3.6.

```
1  ConfDecl       := SystemDecl | TemplateDecl
2  SystemDecl     := "System" Symbol ( "(" ParamList ")" )?
                     "{" SystemBody "}"
3  TemplateDecl   := "SystemTemplate" Symbol ( "(" ParamList ")" )?
                     "{" SystemBody "}"
4  ParamList      := Symbol ( "," Symbol )*
5  SystemBody     := PropertyDecls ElemDecls
6  ElemDecls      := ( InstanceDecl )*
                     ( PipeDecl )*
                     ( IncludeDecl )*
                     ( MappingDecl )*
7  InstanceDecl   := "Instance" Symbol "{" PropertyDecls "}"
8  PipeDecl       := "Pipe" ( Symbol )?
                     "[" ExtParam ( "," ExtParam )+ "]"
                     "{" PropertyDecls "}"
9  IncludeDecl    := "Include" Symbol ":" ExtParam ";"
10 PropertyDecls  := ( PropertyDecl )*
11 PropertyDecl   := Symbol "=" ElemExp ";"
12 ElemExp        := StrValue | IntValue | SetExp | ExtRef
13 SetExp         := "{" ( RangeExp | ElemList )? "}"
14 RangeExp       := IntValue ".." ( IntValue )?
15 ElemList       := ElemExp ( "," ElemExp )*
16 ExtRef         := ExtParam ( "(" ExtParamList ")" )?
17 ExtParamList   := ExtParamElem ( "," ExtParamElem )*
18 ExtParamElem   := ExtParam | StrValue
19 ExtParam       := "#" <IDENTIFIER> ( "." <IDENTIFIER> )*
20 Symbol         := <IDENTIFIER>
21 StrValue       := <STRING_LITERAL>
22 IntValue       := <INTEGER_LITERAL>
```

Figura 3.6. Gramática da notação sem *tags* para descrições de arquitetura em LindaX.

A Figura 3.7 descreve o componente composto V apresentado no Capítulo 2 – e que é reproduzido na figura – em termos de sistemas e *templates*. Os tipos referenciados no exemplo correspondem àqueles descritos na Figura 3.4.

```
1  SystemTemplate IPModule( x,y,z ) {
2
3    Style = #NPUBasedForwarder;
4
5    Instance I1 { Type = #UIOModule( #x ); }
6    Instance I2 { Type = #UIOModule( #x ); }
7    Instance F  { Type = #UForwardingModule( #y ); }
8
9    Pipe pipe1[#I1,#F] { Type = #SLocalEnv( #z ); }
10   Pipe pipe2[#F,#I2] { Type = #SLocalEnv( #z ); }
11
12   Mapping p1: #I1;
13   Mapping p2: #I2;
14 }
15
16 System Router( x,y,z ) {
17   Style = #GenericForwarder;
18
19   Instance V { Type = #IPModule( #x,#y,#z ); }
20   ... //outras ligadas a V.p1 e V.p2...
21 }
```

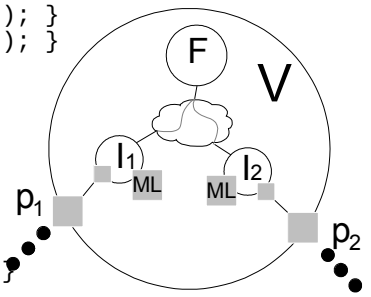


Figura 3.7. Descrição de arquitetura baseada em estilo.

Algumas observações importantes devem ser feitas acerca desse exemplo. Primeiro, os *pipes* são nomeados, para que possam ser devidamente referenciados nas descrições da visão de recursos do modelo (vide Seção 3.4). Em segundo lugar, descrições de sistemas e *templates* também podem ser parametrizadas (a estrutura de parâmetros do esquema `lindaxcnf` é idêntica a do esquema `lindaxtyp`), permitindo um melhor reuso das mesmas. No exemplo, os parâmetros dessas descrições são passados, em última instância, como argumentos para tipos de componentes e *pipes*. Novamente, valores de parâmetros não resolvidos na própria descrição (caso dos parâmetros `x`, `y` e `z` da descrição do sistema `Router`) são geralmente descritos em arquivos de propriedades específicos que alimentam cada ferramenta de refinamento ou síntese distinta.

Pode-se perceber, pelo exemplo, que LindaX apresenta muitas semelhanças em relação às ADLs tradicionais (no Capítulo 6 as sintaxes de algumas dessas ADLs são ilustradas resumidamente). É crucial salientar, no entanto, que a semântica (e mesmo alguns aspectos arquiteturais) associada a descrições em LindaX não pode ser obtida completamente a partir da sintaxe somente. Por exemplo, não fica claro exatamente quais portas dos componentes $\texttt{I}_1$, `F` e $\texttt{I}_2$ e quais pontos de acesso dos *pipes* $\texttt{pipe}_1$ e $\texttt{pipe}_2$ são associados entre si (linhas 8 e 9). Pouco pode ser afirmado também com respeito a quais portas de $\texttt{I}_1$ e $\texttt{I}_2$ são efetivamente acessíveis por elementos externos à configuração (linhas 11 e 12). Objetiva-se neste trabalho que ferramentas de refinamento e síntese para essas descrições sejam capazes não só de precisar os relacionamentos acima, como também inferir a existência de determinados constituintes em uma configuração. É a partir da identificação do estilo seguido pela configuração (linhas 2 e 16) que essas ferramentas definem a DSL LindaX específica com a qual a descrição mantém correspondência e, conseqüentemente, quais ferramentas extensoras detêm conhecimento semântico a respeito dessa descrição. São as ferramentas extensoras que, no exemplo, identificariam exatamente quais portas e pontos de acesso devem ser acoplados. O relacionamento entre a semântica das descrições de arquitetura, os estilos e as ferramentas extensoras próprias de cada estilo é apresentado em detalhe no Capítulo 4.

## 3.4. Visão de recursos (esquema-base `lindaxres`)

O esquema `lindaxres` define um formato para especificação das abstrações da visão de recursos do modelo SCM, a partir dos tipos complexos XML `Task` e `Provider`. Especificações nas visões arquitetural e de recursos são feitas em separado, tornando-as mais simples de serem descritas e compreendidas. As ferramentas de refinamento e síntese são responsáveis por "costurar" essas especificações coerentemente.

Em uma especificação de recursos, cada tarefa e provedor possui um identificador XML próprio e uma lista não vazia de propriedades. A única propriedade obrigatória nessa lista é `Architecture` (tipo XML `List`), que faz referência ao conjunto de componentes e *pipes* que a tarefa ou provedor engloba. Propriedades adicionais podem ser usadas para prover maiores informações acerca desses elementos.

É interessante observar que a visão de recursos criada nesta tese para o modelo SCM, a partir da inclusão do conceito de tarefa[7], é ortogonal àquela da visão arquitetural. Os elementos da visão arquitetural possibilitam a representação da estrutura de um sistema em termos das funções executadas por cada um de seus módulos. Porém, essa visão arquitetural não permite representar adequadamente outros aspectos, tipicamente não-funcionais, que se apresentam 'simultaneamente' em vários módulos, como distribuição, gerência de recursos, QoS, segurança etc. Nesse contexto, há na literatura uma forte tendência no uso de princípios de desenvolvimento de software orientado a aspectos (Kiczales et al., 1997), que sugerem justamente uma ortogonalidade na representação de aspectos funcionais e não-funcionais. Embora sejam potencialmente aplicáveis para representar aspectos não-funcionais variados, tarefas e provedores são usados neste trabalho somente para relacionar a execução de atividades de computação e comunicação a espaços de endereçamento e a partições de recursos computacionais e de comunicação. Outras possíveis aplicações da visão de recursos que podem vir a ser vislumbradas pela linguagem LindaX são discutidas no Capítulo 7.

[7] O conceito de provedor, como discutido na Capítulo 2, já existia no modelo SCM para delimitar escopos dos mais variados tipos, ligados a ambientes de oferecimento de serviços e sua gerência. Neste capítulo, a noção de provedor como um "escopo" é usada para representar também contêineres de recursos de comunicação.

A gramática em EBNF da notação sem *tags* para descrições da visão de recursos em LindaX é apresentada na Figura 3.8.

```
1  ResDecl      := TaskDecl | ProvDecl
2  TaskDecl      := "Task" Symbol ( "(" ParamList ")" )?
                    "{" ExecBody "}"
3  ProvDecl      := "Provider" Symbol ( "(" ParamList ")" )?
                    "{" ExecBody "}"
4  ParamList     := Symbol ( "," Symbol )*
5  ExecBody      := PropertyDecls
6  PropertyDecls := ( PropertyDecl )*
7  PropertyDecl  := Symbol "=" ElemExp ";"
8  ElemExp       := StrVal | IntVal  | SetExp | ExtRef
9  SetExp        := "{" ( RangeExp | ElemList )? "}"
10 RangeExp      := IntVal ".." ( IntVal )?
11 ElemList      := ElemExp ( "," ElemExp )*
12 ExtRef        := ExtParam ( "(" ExtParamList ")" )?
13 ExtParamList  := ExtParamElem ( "," ExtParamElem )*
14 ExtParamElem  := ExtParam | StrVal
15 ExtParam      := <EXTERNAL_IDENTIFIER>
16 Symbol        := <IDENTIFIER>
17 StrVal        := <STRING_LITERAL>
18 IntVal        := <INTEGER_LITERAL>
```

Figura 3.8. Gramática da notação sem *tags* para descrições de recursos em LindaX.

A Figura 3.9 descreve um exemplo de especificação de tarefa relacionado à descrição arquitetural da Figura 3.7. O fato dos módulos de E/S descritos na Figura 3.7 serem ambos passivos implica em uma única atividade de computação sendo exercida desde a entrada até a saída de um pacote no módulo IP do roteador. Uma forma alternativa de descrever essa atividade seria, portanto, incluindo o componente composto `V`, em si, na tarefa. Essa noção de "atividade" depende do ambiente de execução que hospeda o roteador (hardware, sistema operacional, plataforma de programação), podendo se referir a uma única *thread* ou LWP em um PC, um *pool* de *threads* em um processador específico de manipulação de pacotes dentro de uma NPU, entre outros arranjos. Qualquer que seja o ambiente em questão, propriedades específicas podem ser definidas para relacionar a tarefa a partições de recursos ou espaços de endereçamento. A título de exemplo, a propriedade parametrizada `AddrSpace` é definida na Figura 3.9 a fim de permitir identificar a localidade da tarefa. Novamente, o conceito de "localidade" depende do ambiente de execução específico. No caso de um roteador, esse conceito envolverá obviamente o esquema de endereçamento IP, porém pode haver também questões específicas acerca da plataforma de programação. Por exemplo, componentes da plataforma OpenCOM que implementem os módulos do roteador podem ser associados a uma tarefa cuja propriedade `AddrSpace` possui como valor a dupla `{endereço_IP,id_de_cápsula}`. Essa dupla indica que os componentes devem ser instanciados em uma cápsula OpenCOM presente na

máquina hospedeira do roteador. Novamente, todas essas questões acerca do mapeamento da visão de recursos para outras ADLs e plataformas de programação são discutidas no Capítulo 5.

```
1  Task T_F( addr ) {
2    Architecture = {#Router.V.I1,
3                    #Router.V.F,
4                    #Router.V.I2};
5    AddrSpace = #addr;
6  }
```

Figura 3.9. Descrição de atividades de computação e comunicação em LindaX.

## 3.5. Estilos (esquema-base `lindaxsty`)

O esquema `lindaxsty` permite a definição de estilos na linguagem LindaX, a partir do tipo complexo XML `Style`. Um estilo caracteriza um conjunto de arquiteturas que compartilham propriedades estruturais e semânticas. Em LindaX tais propriedades são descritas por meio de um 'vocabulário' e um conjunto de 'restrições'. O vocabulário de entidades de um estilo é composto por tipos descritos a partir do esquema `lindaxtyp`, que é importado por `lindaxsty`. As restrições de um estilo são predicados (tipo complexo XML `Predicate`) que estabelecem regras a serem obedecidas pelas arquiteturas que seguem o estilo. Essas regras podem ser dos mais variados tipos, envolvendo restrições topológicas ou temporais. Nesse sentido, o esquema `lindaxsty` não restringe o uso de uma determinada representação nas restrições; elas podem incluir texto livre (linguagem natural), diferentes classes de lógica etc. Esquemas de extensão de `lindaxsty` permitem a definição de representações específicas, por meio de especializações do tipo XML `Predicate`. Nesta tese, somente duas representações são definidas: texto livre e lógica de primeira ordem (esta última, apresentada na Seção 3.5.1). O Capítulo 7 discute o uso de outras notações possíveis.

A versão atual da gramática em EBNF da notação sem *tags* para descrições de estilos em LindaX é apresentada na Figura 3.10. Note que, com a introdução de novas representações para restrições, a gramática em EBNF da notação sem *tags* para descrições de estilos deve ser alterada apropriadamente.

```
1  StyleDecl    := "Style" Symbol "{" StyleBody "}"
2  StyleBody    := PropDecls VocDecls PredDecls
3  VocDecls     := TypeDecls
4  TypeDecls    := ... (* gramática da Figura 3.2 *)
5  PredDecls    := TxtPredDecl | FolPredDecl
6  TxtPredDecl  := SingleStrVal
7  FolPredDecl  := ... (* gramática da Figura 3.13*)
8  PropDecls    := ( PropDecl )*
9  PropDecl     := Symbol "=" ElemExp ";"
10 ElemExp      := StrVal | IntVal | SetExp | ExtRef
11 SetExp       := "{" ( RangeExp | ElemList )? "}"
12 RangeExp     := IntVal ".." ( IntVal )?
13 ElemList     := ElemExp ( "," ElemExp )*
14 ExtRef       := ExtParam ( "(" ExtParamList ")" )?
15 ExtParamList := ExtParamElem ( "," ExtParamElem )*
16 ExtParamElem := ExtParam | StrVal
17 ExtParam     := <Ext_IDENTIFIER>
18 Symbol       := <IDENTIFIER>
19 StrVal       := <STRING_LITERAL>
20 IntVal       := <INTEGER_LITERAL>
```

Figura 3.10. Gramática da notação sem *tags* para descrições de estilos em LindaX.

O exemplo de módulo IP de roteador apresentado ao longo deste capítulo é retomado nesta seção para ilustrar a aplicabilidade de estilos LindaX. A Figura 3.3 descreve, em linhas gerais, um estilo que modela o plano de dados de um roteador genérico. Conforme definido no predicado da figura (linhas 5 a 14), para estar em conformidade com esse estilo, uma arquitetura de roteador deve ter todos os seus componentes dotados de portas específicas para passagem de pacotes – isto é, portas dos tipos de interface declarados no vocabulário do estilo.

```
1  Style GenericForwarder {
2    InterfaceType PPacketPush( behv ) { Behaviour = #behv; }
3    InterfaceType PPacketPull( behv ) { Behaviour = #behv; }
4
5    FolPredicate PacketInterfaces {
6      forall c: Components(); {
7        exists p: Ports( #c ); {
8          forall s: Signatures( #p ); {
9            [PropertyValue( #s,“Type” ) == #PPacketPush] or
10           [PropertyValue( #s,“Type” ) == #PPacketPull]
11         }
12       }
13     }
14   }
15 }
```

Figura 3.11. Descrição de estilo em LindaX.

Estilos podem também ser definidos como extensões, ou ‘sub-estilos’, de outro estilo. Um sub-estilo inclui todo o vocabulário e conjunto de restrições de seu estilo-pai. Por exemplo, um sub-estilo para roteadores baseados em NPUs pode ser derivado do estilo da Figura 3.3, adicionando-se a ele: (i) tipos para os componentes representando módulos de encaminhamento e de E/S e para os *pipes* de interligação desses módulos, como os da Figura 3.4, e (ii) uma restrição definindo que todo componente de um dos tipos supracitados deve ter sua

execução obrigatoriamente atrelada a um processador específico de manipulação de pacotes dentro da NPU, para garantir um retardo baixo de processamento.

A Figura 3.12 ilustra a descrição do sub-estilo descrito acima. Esse exemplo traz como novidade, em relação ao exemplo anterior, o fato de que predicados também podem ser parametrizados. Novamente, essa facilidade propicia um melhor reuso de estilos. No exemplo, ela permite a descrição em separado da localidade de uma tarefa, que é uma informação tipicamente dependente do ambiente de execução. Ambos os estilos ilustrados nesta seção são versões modificadas de estilos com os quais a DSL LindaRouter mantém correspondência.[8] A versão original desses estilos é apresentada no Capítulo 4.

```
1  Style NPUBasedForwarder {
2    Superstyle = #GenericForwarder;
3
4    ComponentType UIOModule( impl,behv ) { ... }
5    ComponentType UForwardingModule( impl,behv,nports ) { ... }
6    PipeType SLocalEnv( impl,behv ) { ... }
7
8    FolPredicate FastPath( addr ) {
9      forall t: Tasks( Components( #UIOModule ) union
10                      Components( #UForwardingModule ) ); {
11       [PropertyValue( #t,“Address” ) == #addr]
12     }
13   }
14 }
```

Figura 3.12. Descrição de sub-estilo em LindaX.

### 3.5.1. Lógica de predicados (esquema de extensão `lindaxfolpredicate`)

O esquema de extensão `lindaxfolpredicate` é definido a partir do esquema-base `lindaxsty` para permitir a especificação de restrições topológicas em um estilo e a verificação dessas restrições sobre as configurações que seguem esse estilo. Esse esquema define uma sintaxe para predicados em lógica de primeira ordem (*First-Order Logic* – FOL). que opera sobre o vocabulário do estilo e aquelas entidades que constituem a configuração que se quer restringir. Isso inclui também referências a tarefas e provedores. A lógica FOL definida por esse esquema (tipo complexo XML `FolPredicate`, herdado de `lindaxstyle:Predicate`) oferece os operadores lógicos convencionais em uma lógica de primeira ordem: `and`, `or`, `not`, `forall` e `exists`.[9] Além disso, essa lógica apresenta os elementos listados a seguir:

[8] O objetivo dessas modificações foi tornar o mais simples possível os exemplos apresentados nesta seção e, ao mesmo tempo, realçar características da linguagem não exploradas pela versão original desses estilos.

- **Domínios**: a lógica FOL oferece cadeias de caracteres (tipo XML `string`), números (tipo XML `hexBinary`), conjuntos ordenados (tipo complexo XML `Set`, definido em `lindaxfolpredicate`) e referências (tipo XML `ExternalRef`) como valores possíveis para símbolos e operandos em geral. Conjuntos ordenados são conjuntos cujos elementos podem ser indexados como em um *array*. Referências apontam para os identificadores XML de tipos descritos em estilos, bem como para componentes, *pipes,* tarefas e provedores em configurações;
- **Símbolos**: a lógica FOL permite a definição de símbolos para constantes e variáveis dentro dos domínios de valores acima. Referências e cadeias de caracteres são sempre armazenadas em constantes. Números e conjuntos ordenados podem ser armazenados em constantes ou variáveis. Modificações nos valores de variáveis estão amarradas aos operadores de quantificação universal (`forall`) e existencial (`exists`).
- **Operadores e funções**: a lógica FOL oferece operadores e permite a definição de símbolos para funções. Operadores e funções mapeiam conjuntos de constantes e variáveis em valores dentro de um dos domínios citados acima. Os seguintes operadores são oferecidos pela lógica FOL:
    a) $N_1$ `+` $N_2$, $N_1$ `-` $N_2$, $N_1$ `*` $N_2$ e $N_1$ `/` $N_2$: mapeiam uma dupla de números {$N_1$,$N_2$} em outro número;
    b) $S_1$ `union` $S_2$, $S_1$ `intersection` $S_2$ e $S_1$ `minus` $S_2$: mapeiam uma dupla de conjuntos {$S_1$,$S_2$} em outro conjunto;

    As seguintes funções são oferecidas pela lógica FOL:
    a) `At(S,N)`: mapeia um conjunto ordenado `S` e um número (inteiro) `N` em um valor dentro dos domínios definidos pela lógica. Esse valor corresponde ao elemento do conjunto `S` na posição indicada pelo número `N`;
    b) `Cardinality(S)`: mapeia um conjunto ordenado `S` em um número que corresponde à cardinalidade do conjunto;

[9] A lógica FOL definida pelo esquema `lindaxfolpredicate` não define o operador `if X Y` comumente encontrado em outras linguagens baseadas em lógica de primeira ordem. Contudo, esse operador pode ser igualmente representado na lógica FOL pela expressão lógica `not X or Y`.

c) `PowerSet(S)`: mapeia um conjunto ordenado `S` em outro que representa o *powerset* (conjunto de todos os subconjuntos) de `S`. Nesse caso, permutações de um subconjunto não são consideradas elementos distintos do *powerset*;

d) `SequenceSet(S)`: similar a `PowerSet`, exceto pelo fato de permutações de um conjunto serem consideradas elementos distintos do *powerset*;

e) `PropertyValue(#ref,C)`: mapeia uma referência `#ref` e uma cadeia de caracteres `C` em um valor dentro dos domínios definidos pela lógica. Esse valor corresponde àquele da propriedade de nome `C` do elemento referenciado por `#ref` na configuração ou no vocabulário do estilo;

f) `Components(#ref)`: mapeia uma referência `#ref` a um tipo de componente em um conjunto de referências a instâncias desse tipo presentes na configuração. Se a referência ao tipo for omitida, é retornado o conjunto de referências a todos os componentes da configuração;

g) `Pipes(#ref)`: mapeia uma referência `#ref` a um tipo de *pipe* em um conjunto de referências a instâncias desse tipo presentes na configuração. Se a referência ao tipo for omitida, é retornado o conjunto de referências a todos os *pipes* da configuração;

h) `Pipe(#ref_C1,#ref_P1,#ref_C2,#ref_P2)`: mapeia uma quádrupla contendo referência a dois componentes e a duas portas desses componentes em uma referência ao *pipe* existente entre os dois componentes, por meio das portas indicadas;

i) `Ports(#ref)` e `AccessPoints(#ref)`: mapeiam uma referência `#ref` a uma instância (componente ou *pipe*) em um conjunto de referências aos pontos de interação dessas instâncias;

j) `Signatures(#ref)`: mapeia uma referência `#ref` a uma porta ou ponto de acesso em um conjunto de referências a assinaturas desses pontos de interação;

k) `Tasks(S)`: mapeia um conjunto `S` de referências a componentes em um conjunto de identificadores de tarefas associadas a esses componentes;

l) `Providers(S)`: mapeia um conjunto `S` de referências a *pipes* em um conjunto de identificadores de provedores associados a esses componentes;

- **Operadores de predicado**: a lógica FOL define operadores de predicado que mapeiam conjuntos de símbolos quaisquer (constantes, variáveis e valores de retorno de funções) em valores-verdade (tipo nativo XML `boolean`). Esses valores são usados como operandos dos operadores lógicos da lógica FOL. Os operadores de predicado oferecidos são: `==`, `!=`, `>`, `>=`, `<`, `<=` e `in`.

A gramática em EBNF da notação sem *tags* para a lógica FOL é apresentada na Figura 3.13. A Figura 3.14 identifica esses elementos no contexto de um exemplo.

```
1  FolPredDecl:= LogicalExp
2  LogicalExp := "forall" FParamsExp "{" LogicalExp "}"
                 | "exists" FParamsExp "{" LogicalExp "}"
                 | BoolExp
3  FParamsExp := FParam ( FParamsExp )*
4  FParam     := Symbol ":" FolSetExp ( "where" BoolExp )? ";"
5  BoolExp    := OrExp
6  OrExp      := AndExp ( "or" AndExp )*
7  AndExp     := BoolUnExp ( "and" BoolUnExp )*
8  BoolUnExp  := "(" BoolExp ")" | "not" BoolExp | "[" CmpExp "]"
9  CmpExp     := FolElemExp ( "==" FolElemExp
                             | "!=" FolElemExp
                             | ">=" FolElemExp
                             | "<=" FolElemExp
                             | ">" FolElemExp
                             | "<" FolElemExp
                             | "in" FolSetExp )
10 FolElemExp := StrVal | ExtParam | FuncExp | FolSetExp | IntExp
11 IntExp     := AddExp
12 AddExp     := MulExp ( "+" MulExp | "-" MulExp )*
13 MulExp     := IntUnExp ( "*" IntUnExp | "/" IntUnExp )*
14 IntUnExp   := "(" AddExp ")" | "val" ( FuncExp | ExtParam ) | IntVal
15 FolSetExp  := UnionExp
16 UnionExp   := InterExp ( "union" InterExp | "minus" InterExp )*
17 InterExp   := SetUnExp ( "intersection" SetUnExp )*
18 SetUnExp   := "(" UnionExp ")" | "set" ( FuncExp | ExtParam ) | SetVal
19 SetVal     := "{" ( FolRangeExp | FolElemList )? "}"
20 FolRangeExp:= IntExp ".." ( IntExp )?
21 FolElemList:= FolElemExp ( "," FolElemExp )*
22 FuncExp    := Symbol "(" ( ParamList )? ")"
23 ParamList  := FolElemExp ( "," FolElemExp )*
24 ExtParam   := "#" <IDENTIFIER> ( "." <IDENTIFIER> )*
25 Symbol     := <IDENTIFIER>
26 StrVal     := <STRING_LITERAL>
27 IntVal     := <INTEGER_LITERAL>
```

Figura 3.13. Gramática da notação sem *tags* para a lógica FOL.

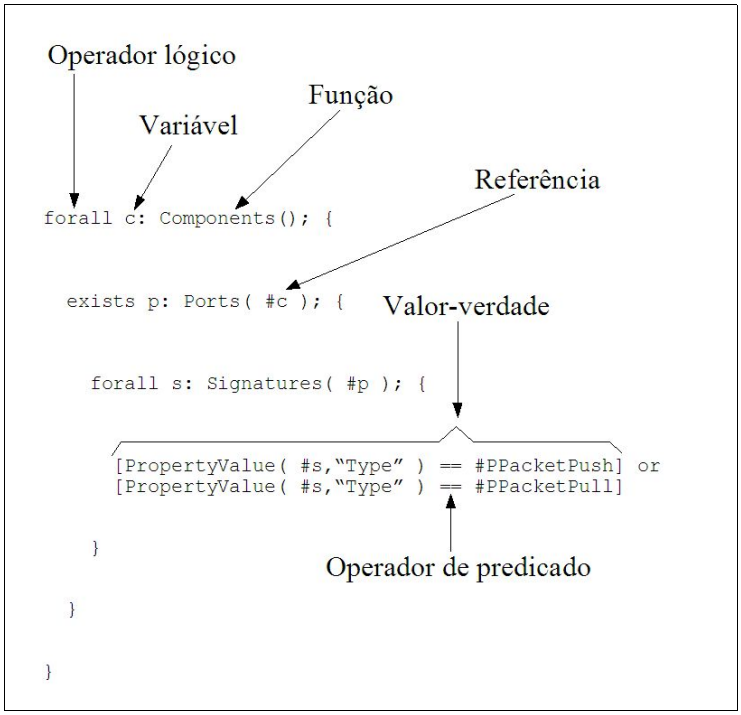


Figura 3.14. Identificação dos elementos da lógica FOL.

## 3.6. Sumário

Este capítulo apresentou a estrutura da linguagem de especificação LindaX, que é a principal contribuição da tese presente. O principal aspecto inovador de LindaX é o uso combinado de características de ADLs – em particular, a noção de estilos – com um arcabouço sintático único para DSLs com semânticas distintas. Nessa combinação, é fundamental o papel das ferramentas extensoras, que detêm o conhecimento da semântica de cada DSL, como será visto no Capítulo 5. Para ilustrar esses conceitos, no capítulo seguinte duas DSLs especializadas a partir do arcabouço sintático de LindaX são apresentadas.