MARCELO JACCOUD AMARAL

Mantendo código-fonte em XML – rumo ao hiperfonte

Um estudo sobre o uso de linguagens de marcação no desenvolvimento e manutenção de programas

Dissertação apresentada ao Departamento de Informática da PUC-Rio como parte dos requisitos para a obtenção do título de Mestre em Ciências de Informática

Orientador: Prof. Edward Hermann Haeusler

Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro

Rio de Janeiro, 29 de maio de 2001

Para Agnes, lux mea.

Agradecimentos

A Deus, por mais esta oportunidade de crescer.

À minha esposa Agnes, pelo apoio emocional incondicional e constante.

À minha família, que sempre me incentivou a aprender.

Ao meu orientador, Edward Hermman, pela amizade e contínua disposição em ajudar.

À PETROBRAS, pelo suporte financeiro, e aos colegas de trabalho, pelas muitas idéias e colaborações.

Aos professores do Departamento de Informática, pela maneira exemplar de seduzir-nos com o conhecimento e motivar-nos à busca contínua de novos horizontes.

Aos colegas do TecMF, pelo ambiente saudável e valiosas dicas.

Resumo

O formato de arquivo em que hoje se mantém código-fonte dificulta tanto a edição manual quanto a automática. Usando a linguagem de marcação XML *(Extensible Markup Language)* e um modelo de documento apropriado à linguagem de programação, o código-fonte pode ser transformado num hiperfonte que apresenta várias vantagens em relação ao texto comum. São apresentados métodos para modelar os documentos XML, criar os programas tradutores que importam código no formato tradicional (usando TXL) e outras ferramentas que operam sobre o hiperfonte. São fornecidos modelos de documento para código-fonte em TXL, fortran e C++, acompanhados dos programas de importação, folhas de estilo, e vários exemplos de transformações escritas em XSLT, uma linguagem projetada para transformar documentos XML.

Abstract

The file format used today to maintain source code hampers both manual and automatic edition. Using the Extensible Markup Language (XML) and an appropriate document model designed for the programming language, the source code can be turned into a hypersource which exhibits several advantages over the plain text version. We present methods for designing the XML document models, creating translation programs to import traditional source code (using TXL), and building other tools to work with the hypersource. Examples of document models for TXL, Fortran and C++ are given, together with the import programs, style sheets and several transformations written in XSLT, a language designed to transform XML files.

SUMÁRIO

ROL DE ILUSTRAÇÕES

ROL DE LISTAGENS

```
int main(void) {
/* This is a program. */
return 0;
}
```

1. INTRODUÇÃO

Desde que as primeiras linguagens de programação foram desenvolvidas, houve uma preocupação de dotá-las de mecanismos que permitissem documentar o código. Comentários explicativos ainda são indispensáveis em virtude da grande distância que existe entre as linguagens naturais e as de programação. Enquanto o vernáculo abunda em versatilidade, permitindo que nele se redijam todo o tipo de textos, da norma técnica à mais rebuscada poesia, as linguagens de programação primam pela rigidez de gramática e escopo — até hoje, estão muito mais próximas da máquina do que do ser humano que as usa. Para expressar um algoritmo com clareza, um programador tem a seu dispor linguagens com sintaxe e semântica muito bem definidas, porém limitadas em sua expressividade pela própria capacidade de processamento do computador. Como os custos de manutenção estão diretamente relacionados à capacidade dos técnicos de entender como um sistema funciona e quais suas limitações, não é de se espantar que um código de qualidade, por mais auto-explicativo que seja, venha sempre adornado de comentários.

Ou melhor... deveria vir. Apesar de haver um consenso no que tange à necessidade de documentar o fonte apropriadamente, a maior parte do código à nossa

disposição ainda possui pouca ou nenhuma documentação, seja ela embarcada (incluída no próprio código) ou anexada (em outros arquivos). Quando existe, não é incomum que esteja fora de sincronia com a versão atual dos arquivos ou incompleta. Num mundo onde o tempo adicional gasto na documentação do código é comumente vítima de cronogramas irreais e freqüentemente não há sequer um processo controlado de produção, nos deparamos com uma abundância de código pouco legível e de difícil manutenção. A situação é ainda pior quando se trata de sistemas legados: na maioria das vezes, nenhum dos programadores originais do sistema está disponível, e a dificuldade em compreender a arquitetura e a implementação do programa é tão grande que às vezes é mais econômico reescrevê-lo.

Esta doença crônica que aflige os programas de computador vem sendo atacada de vários ângulos. No ensino, através da ênfase na utilização das boas práticas de programação. No projeto de linguagens, aumentando o poder expressivo e de abstração, de forma que o código fique mais conciso e claro. Criam-se ferramentas de apoio que direcionem o programador a documentar o código. E os engenheiros de *software* estão sempre a lembrar-nos da importância de estabelecer procedimentos metódicos de verificação, para assegurar a consistência da documentação. A maioria das soluções esbarra no grande número de artefatos gerados durante a vida de um sistema e na grande variedade de formato, muitas vezes proprietários, que atrapalham a verificação automática de consistência.

Entretanto, até hoje, o fonte de um programa evoluiu muito pouco no seu formato básico. Dos cartões perfurados que conviviam com as primeiras versões de fortran[1], onde cada coluna tinha um significado, passando pelo formato livre do

---

[1] A ortografia portuguesa dita que os glossônimos são grafados com inicial minúscula, ao contrário do inglês, que usa capitulares. Compare: "*I speak English*", mas: "Eu falo português". Procuramos evitar o anglicismo, inclusive para os acrônimos (fortran, cobol). Tal não se aplica às siglas: C, XML.

pascal e chegando a uma especificação em Z, onde a endentação das linhas possui semântica, o fonte ainda é um arquivo-texto dos mais simples. Se há hoje editores sensíveis a alguns aspectos da sintaxe, é porque tal capacidade foi embutida nos próprios editores. Qualquer programa que trate o código como algo além duma seqüência aleatória de caracteres precisa analisá-lo sintaticamente, i.e., necessita conhecer seu léxico e sua sintaxe. Quanto mais esperta e inteligente a ferramenta, mais conhecimento necessita, e mais complexo e caro será seu desenvolvimento. Um editor com "coloração sintática" conhece apenas algumas regras de formação de *tokens*; um bom compilador deve verificar todas as restrições sintáticas e semânticas da linguagem.

Com a explosão da Internet, o aumento vertiginoso da troca de informações, e a globalização dos serviços, um grande número de padrões foram desenvolvidos para normatizar formatos. Um dos mais significativos esforços foi feito pelo consórcio Unicode, que padronizou a codificação de texto em dezenas de alfabetos, dando origem ao primeiro padrão realmente universal para informação escrita. Paralelamente, uma família de normas derivadas do padrão ISO 8879:1986 (SGML) permite que possamos codificar informação textual hierarquicamente, através do uso de marcadores. Diversas aplicações têm se baseado nesse padrão, normatizando a troca de informações dos mais variados tipos, desde dados contábeis até gráficos vetoriais complexos.

O que nos leva a repensar o código-fonte. Não seria a hora do velho arquivo-texto ser substituído por um esquema de codificação mais inteligente? Neste trabalho pretendemos mostrar que é possível manter código-fonte usando XML e seus padrões periféricos, e que tal esquema, além de permitir uma documentação muito mais rica e universalizada, facilita e barateia o desenvolvimento de ferramentas para tratamento mecanizado do código, desde simples formatadores a sofisticados verificadores de padrões.

Nossa contribuição

Com este trabalho mostramos que o uso de XML na codificação (em oposição a formatos fechados) estabelece uma plataforma versátil e universal tanto para a criação e manutenção do código-fonte propriamente dito, pois permite desenvolver rapidamente ferramentas de automação sofisticadas, como para tratar os comentários usualmente inclusos nos programas. Ao segregar os comentários e caracterizá-los como metainformação, removemos uma série de restrições que nos permitem atingir novos patamares de qualidade na documentação dos programas.

Na seção 2, vemos como evoluiu a codificação dos fontes de programas e quais as vertentes mais importantes.

Na seção 3, revisamos os padrões de marcação existentes e sua aplicabilidade à codificação dos fontes.

Nas seções 4 e 5, analisamos as opções de codificação usando XML e sugerimos uma solução pragmática, ao redor da qual mostramos como construir vários tipos de transformadores.

Na seção 6, alguns exemplos concretos de transformações são exibidas, e sua implementações discutidas com mais detalhe.

Na seção 7, ponderamos sobre os resultados desta pesquisa e sugerimos rumos para desenvolvimentos ulteriores.

OBSERVAÇÃO: Como o texto aborda várias linguagens, alguns termos podem ter interpretações duvidosas. Em particular, a terminologia usada no contexto das linguagens de marcação não está bem consolidada, sobretudo em português. Um glossário foi incluído ao fim da dissertação para dirimir dúvidas e esclarecer alguns conceitos, e espera-se que ele não apenas ajude a leitura, mas contribua para a sedimentação dos termos na comunidade lusófona. O autor pede desculpas por algum portuguesismo.

```
int main ( void )
{
 Das ist ein Programm.
return 0;
}
```

2. HISTÓRICO

Nesta seção, reveremos a evolução do código-fonte sob vários aspectos, analisando como cada novidade afetou o modus operandi dos programadores, tanto na comunidade acadêmica como na indústria de *software*. Arrolaremos durante o processo os sistemas e padrões mais relevantes.

2.1. Codificação

Desde que a codificação EBCDIC, usada nas perfuradoras de cartões e depois herdada pelos *mainframes*, foi preterida em prol do ASCII, pouco se avançou na forma de armazenar o código-fonte. A simplicidade deste padrão, em que cada caráter é representado por apenas 7 bits, pode facilitar a portabilidade do texto entre diferentes plataformas, mas o conjunto extremamente reduzido de caracteres mal atende aos usuários de origem anglo-saxônica.

Esse atraso tem explicação. A comunicação universal de texto em forma digital é um problema muito mais complexo do que a maioria das pessoas — incluindo muitos profissionais de informática — imagina. E isso ocorre porque texto é algo intrinsecamente complicado: os diferentes sistemas de escrita usados pelo homem usam várias miríades de caracteres e empregam diferentes preceituários sobre como combiná-los de forma oral ou escrita. Conseqüentemente, a tarefa de criar uma representação digital de caracteres e texto precisa satisfazer restrições técnicas, culturais e lingüísticas.

As soluções vinham sendo desenvolvidas de forma localizada, gerando muitos esquemas de codificação de caracteres diferentes (e mutuamente incompatíveis), cada qual otimizado para um idioma ou sistema de escrita. Um exemplo típico são as codificações definidas pela norma ISO-8859. Há variantes que contemplam desde os alfabetos latinos do Oeste Europeu (ISO-8859-1) até o cirílico (ISO-8859-5) ou hebraico (ISO-8859-8). A ISO-8859-1 é a codificação mais usada no mundo, e é mais conhecida pelo epíteto de *Latin-1*. Existem também vários padrões para codificação de ideogramas, como ISO-2022-JP (japonês), GB2312 (chinês) e EUC-KR (coreano).

Nos últimos quinze anos, porém, com o impulso do processo de globalização da economia, o problema da codificação recebeu atenção especial, resultando na especificação de dois conjuntos de caracteres universais, formalmente conhecidos como ISO/IEC 10646:1993, desenvolvido pela Organização Internacional de Padronização, e Unicode 2.0, desenvolvido pelo Consórcio Unicode. Fortuitamente, ambas as organizações chegaram à conclusão que não seria prático ou sensato manter dois conjuntos universais de caracteres, e os padrões foram fundidos. Para todos os aspectos práticos, ambos os padrões definem os mesmos caracteres, com a mesma codificação numérica. O padrão Unicode, atualmente na versão 3.0, vem sendo adotado de forma lenta porém progressiva. Os sistemas operacionais de última

geração já possuem suporte intrínseco ao Unicode, o que representa para eles um diferencial importante no mercado global.

Tais avanços, entretanto, não atingiram as ferramentas de desenvolvimento de programas. A razão é simples: a complexidade inerente ao Unicode não se justifica quando tratamos de programas-fontes. A sintaxe e vocabulário usados para representar os elementos de um algoritmo, seja qual for a linguagem de programação escolhida, é muito mais simples que qualquer linguagem natural. O código-fonte em si está, via de regra, limitado ao uso de um conjunto mínimo de caracteres, correspondente ao padrão ASCII. Como os comentários são sumariamente ignorados, a maioria dos compiladores tolera o uso de outras codificações sem dar-lhes maior atenção. Se alguma ferramenta todavia quiser processar os comentários, encontra problemas, pois as linguagens de programação não costumam definir uma maneira padronizada de identificar a codificação usada. Tais ferramentas devem ser configuradas de antemão e é impossível usar alfabetos diferentes no mesmo arquivo.

Algumas linguagens mais recentes, notadamente java [Sun] e C++ [ISO-C++], permitem o uso de caracteres Unicode tanto nos comentários quanto nos nomes de elementos sintáticos. Entretanto, não há notícia de editores ou compiladores que implementem este recurso da linguagem. Destarte, apesar da linguagem permitir que se defina uma constante chamada π (é assim que ela é conhecida em todo o planeta), somos obrigados a usar uma palavra latina: `pi`.[2] A opção de usar as seqüências de escape previstas pela sintaxe (em C++, π pode ser escrito `\u03c0`) é muito pouco atrativa, pois o resultado, apesar de sintaticamente válido e processável, é ilegível para um ser humano normal.

---

[2] Observe-se a metonímia: o nome da letra é usada em seu lugar (a constante é π e não pi). A diferença, aparentemente sutil, tem um grande impacto cognitivo. Compare x/y com xis/ípsilon. Ou ainda: Σ, sigma, somatório.

Pode-se argumentar que a aceitação pela sintaxe de uma linguagem de programação de caracteres não cobertos pelo ASCII é uma característica supérflua. Este ponto de vista é difícil de manter se lembrarmos que a parcela anglófona dos usuários potenciais é pequena, mesmo incluindo os bilíngües, e o uso dos sistemas de escrita nativos aumenta enormemente a legibilidade do código. Além disso, muitos fontes, especialmente os de caráter científico, representam a transcrição de algoritmos que usavam anteriormente algum outro tipo de notação. Comparem-se a legibilidade e a familiaridade da notação ∂x^2 com a da equivalente fortran `DRONDX**2` ou a mesma expressão em C++: `pot(drond_x,2)`.

2.2. Formatação

É raro o grupo de desenvolvimento que não possua recomendações para a formatação do código-fonte. Os mais maduros e organizados chegam a normatizar cada detalhe, incluindo regras para a criação de nomes, endentação, comentários, etc. Um bom exemplo pode ser encontrado em [Staa-00]. O posicionamento e conteúdo dos comentários é um aspecto importante de tais normas, e imprescindível para viabilizar o desenvolvimento de ferramentas de apoio. A conformidade é assegurada implantando-se um mecanismo de verificação do código, na maioria das vezes pouco automatizado e portanto mais susceptível a falhas. Não é sem razão que muitos desenvolvedores reclamam de tais normas, já que seu cumprimento envolve um esforço adicional do qual só lucram mais tarde. Para evitar exageros, [McConnell] recomenda prudentemente que tais normas não ultrapassem 25 páginas, porque a tentativa de normatizar cada pequeno detalhe mostra-se quase sempre infrutífera: os desenvolvedores não se lembrarão de cada detalhe da norma e não conseguirão segui-la. Uma estratégia mais viável é regularizar os aspectos mais importantes, como a escolha de nomes e obrigação de comentar certos itens, e deixar detalhes de

formatação a cargo de ferramentas, assim como a validação de limites no tamanho de nomes, complexidade de rotinas, aninhamento de estruturas, etc.

Um tipo de ferramenta que sempre foi muito popular são os chamados *"pretty-printers"*. Os primeiros destes programas simplesmente geravam uma cópia impressa do código, realçando palavras reservadas e comentários através de variações das fontes disponíveis na impressora. A sofisticação destas ferramentas acompanhou o aumento dos recursos de impressão disponíveis. Hoje, usando impressão a *laser*, é possível produzir cópias que fazem uso de sofisticados recursos de tipografia e composição, e o complementam através de sumários e índices gerados automaticamente a partir do fonte. Um exemplo primoroso deste tipo de ferramenta é SEE [Baecker].

Mais recentemente, há experiências na conversão do código em hipertexto [Amaral-00]. O uso de HTML permite incluir vínculos ativos tanto intra- como interfonte, o que ajuda na tarefa de vasculhar e garimpar código legado, onde há carência ou ausência de comentários. Esta técnica é especialmente útil em linguagens como fortran-77, onde a vinculação é sempre estática e o nível de sobrecarga dos nomes é pequeno ou nulo(3). Em linguagens orientadas a objeto e outras onde a vinculação dinâmica predomina, um *hyperlink* teria que indicar todos os destinos possíveis em tempo de execução, o que torna o hipertexto de pouca utilidade para navegar pelo código.

Todas essas ferramentas de formatação são intrinsecamente de mão única: o material impresso ou hipertexto gerado não é editável, e o fonte é mantido no estado original. A maior legibilidade e facilidade de navegação não estão disponíveis

(3) Em fortran-77, a sobrecarga de nomes só ocorre em funções intrínsecas, como `log`.

durante o processo de edição e alteração do código, e é preciso regenerar periodicamente a documentação para que reflita as modificações introduzidas. A impressão do material implica em um alto custo, e em virtude disso listagens em papel só são geradas eventualmente no fechamento de versões. No caso muito comum do sistema estar em constante manutenção e crescimento, é quase inevitável que tal documentação impressa acabe sem uso, já que nunca está atualizada, exceto para as versões de produção. Se o formato final é eletrônico, o custo de regenerar o hipertexto não é crítico, mas mantê-lo sincronizado pode ser difícil ou impossível se há vários desenvolvedores alterando o código simultaneamente — o hipertexto é, via de regra, gerado somente para uma versão comum que reside no sistema de controle de versão, e não inclui modificações locais feitas por este ou aquele programador.

2.3. Extração de comentários

Na última década, um tipo particular de formatador de programas ganhou enorme popularidade: os extratores de comentários. Tais programas analisam sintaticamente o código, extraindo, junto com a informação semântica necessária para a construção de índices e diagramas, o texto dos comentários, que são associados aos elementos sintáticos relevantes. Essa associação pode ser feita manualmente, inserindo nos comentários marcadores que indicam o elemento sintático ao qual se referem, ou automaticamente, usando regras heurísticas que levam em conta a posição do bloco de comentários em relação aos elementos. Como essa última abordagem, embora menos custosa para o desenvolvedor, pode levar a ambigüidades ou associações errôneas, é comum vir acompanhada de normas rígidas que regem o posicionamento dos comentários.

Os extratores de comentários usam, via de regra, analisadores léxico-sintáticos desenvolvidos especialmente para a linguagem-alvo. Por força do mercado, a maioria está voltada para C, C++ ou java, que são as linguagens de uso mais geral. Java é um caso especial, pois suas versões de distribuição(4) mais recentes incluem uma ferramenta deste tipo, denominada **javadoc**, que se tornou bastante popular. Esta ferramenta exige que os comentários a extrair sejam identificados (usam a seqüência inicial `/**` ao invés de um simples `/*`) e define um conjunto extenso de marcadores destinados a classificar a informação dos comentários. Por exemplo, `@return` indica o início da descrição do valor de retorno de uma função. O resultado final é uma documentação de referência do código cujo grau de detalhe é proporcional ao esforço despendido na escrita destes comentários.

Experiências foram feitas usando ferramentas genéricas, mostrando que é possível obter um alto nível de integração independentemente da linguagem alvo. Um exemplo é Documentu [Braga], uma ferramenta que integra um banco de dados CASE, Talisman [Staa], e uma ferramenta transformacional, Draco-PUC [Leite]. Com Documentu, o texto dos comentários é marcado extensivamente, e as marcações podem ser interpretadas como requisitos ao invés de simples descrições. Uma vez incorporado à base de dados, é possível realizar verificações de consistência e outras operações sobre o código, além de gerar a documentação de referência na forma de hipertexto.

---

(4) Java é uma linguagem desenvolvida pela Sun Microsystems, que detém sua patente. Como sua definição é completamente controlada pela Sun, e não por um padrão internacional, identifica-se a linguagem pelo número da versão de distribuição.

2.4. Ferramentas CASE

Muitas ferramentas CASE (*Computer Aided Software Engineering*) utilizam-se de esquemas de documentação paralelos. O sistema, decomposto em elementos lógicos modulares, é armazenado num banco de dados com uma estrutura otimizada. Nestes casos, o código-fonte é visto como um subproduto destinado à compilação e não à edição e manutenção do sistema. Note-se que não há equivalência entre as duas formas do sistema (a armazenada no banco de dados e a descrita nos fontes). Via de regra, o banco de dados contém uma grande quantidade de informações, relativas à arquitetura e projeto do sistema, que não se encontram no código-fonte. Mesmo nos sistemas que admitem a incorporação de modificações do código ao modelo do banco de dados, isto não é feito de forma completa ou automática. Em particular, comentários inseridos no código se perdem quando este é novamente gerado pela ferramenta. Nestas ferramentas, a documentação de referência é sempre gerada a partir do banco de dados do sistema, e não do código fonte, que pode, inclusive, ser gerado sem comentários.

Estes sistemas, reportadamente eficientes na fase de projeto e desenvolvimento inicial do sistema, não são tão ágeis quando o produto é entregue para uso e entra em fase de manutenção. De face com um defeito, a depuração se dá usualmente através do código-fonte, e o processo de incorporação das correções ao modelo original nem sempre é levado a cabo. Os profissionais encarregados de manter o sistema funcionando quase nunca dispõem do tempo necessário para corrigir o modelo, revalidá-lo, regenerar a documentação, etc. O resultado é um descompasso entre o código-fonte de produção e o modelo original do programa, o que é extremamente danoso e tende a acelerar a degradação natural que um sistema sofre ao longo de sua vida útil.

Talvez o maior empecilho à disseminação generalizada de ferramentas CASE, junto com seu alto custo, é o potencial perigo de armazenar-se partes do sistema em dois formatos diferentes: o banco de dados, usado pela ferramenta, e o código-fonte, necessário para gerar o código-objeto. Este problema não existiria se compiladores e ferramentas CASE utilizassem um mesmo formato para armazenar o fonte: só haveria uma única cópia válida e atualizada da informação. Novamente, há um conflito de interesses, pois a ferramenta CASE possui requisitos lingüísticos diferentes do compilador: enquanto aquela deseja manter diagramas e outros objetos num altíssimo nível de abstração semântica, o compilador está interessado apenas num conjunto de lexemas válidos, que possam ser traduzidos em código-objeto. Estes requisitos implicam em linguagens muito diferentes, como UML e C++, mas certamente não precisariam utilizar meios físicos de armazenamento ou codificações distintas.

Há exemplos que demonstram que tal abordagem seria muito produtiva. O ambiente de desenvolvimento Visual Age, da IBM, mantém num banco de dados o código-objeto e referências para o código-fonte. Quando o fonte é alterado, somente a parte realmente necessária é recompilada, o que aumenta enormemente a eficiência da montagem do programa-objeto.

2.5. Programação letrada

Uma das propostas mais radicais e influentes no âmbito da documentação de programas foi o conceito de *programação letrada* proposto por Knuth. Em suas próprias palavras:

> "I believe that the time is ripe for significantly better documentation of programs, and that we can achieve this by considering programs to be *works of literature*. Hence, my title: 'Literate programming'.
>
> Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.
>
> The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style... He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding using a mixture of formal and informal methods that nicely reinforce each other." [Knuth-84]

Segundo sua visão, o texto de um programa deve ser um trabalho didático que guie o leitor pelo código, como uma obra literária. Já que a ordem de explanação é comumente diferente da ordem dos elementos requerida pela gramática da linguagem de programação, seu sistema, denominado WEB, inclui um mecanismo de macros que permite ao programador/escritor organizar os elementos de forma mais didática.

No sistema original de Knuth, o amálgama de programa e documentação constitui um único arquivo. Nele, a narrativa é entremeada de fragmentos de código que mostram como implementar os procedimentos descritos. A narrativa inclui trechos que descrevem como os procedimentos devem ser encadeados, acompanhados dos fragmentos de código (neste caso, seqüências de macros) que compõem trechos em diferentes níveis de abstração, culminando num trecho que descreve o

programa completo. O fonte resultante é comparado a uma trama (o termo inglês é *web*, que dá nome ao sistema), onde código-fonte e narrativa são tecidos para formar uma peça. Tal código não é compilável, nem é facilmente legível, pois consiste nas definições das macros, que são tão complexas quanto a estrutura original do programa. Para produzir o resultado final (código compilável + documentação), o fonte letrado é submetido a dois processos distintos: a tecedura (ingl. *weaving*), que consiste em gerar o arquivo final de documentação, e o enleio (ingl. *tangling*), que é a produção de código compilável na linguagem de programação escolhida. Estes termos foram cunhados por Knuth no desenvolvimento do sistema WEB, que a tece a documentação em TEX e enleia código pascal. A documentação gerada é extremamente minuciosa, repleta de índices e tabelas que ajudam a navegação pelo código, indicando onde cada termo é definido e referenciado.

> [Brown-Childs] apontam três principais recompensas do uso de WEB:
>
> - ele estimula a organização do código em bases psicológicas ao invés de sintáticas,
> - torna a macroestrutura do programa mais facilmente visível, e
> - encoraja um estilo de escrita exploratório, que induz à consideração cuidadosa dos detalhes do programa.

O uso desta abordagem modular se mostrou útil na redução de erros, como reportado por [Thimbleby]. Entretanto, a alegação que o código letrado estimula um maior reúso em comparação a programação tradicional não foi comprovada [Childs].

Mais tarde, o sistema foi adaptado por [Knuth-Levy], recebendo o nome de CWEB, para trabalhar com C, C++ e Java. Outra adaptação por [Krommes], chamada FWEB, também aceita código em Fortran-77 e RatFor. Todos esses sistemas, apesar de multilíngües, são dependentes da linguagem: para gerar índices e referências

cruzadas eficientemente, é necessário analisar sintaticamente os trechos de código. Também são bastante complexos, o que dificultou sua difusão.

[Ramsey], autor do sistema NOWEB, advoga uma simplificação drástica de tais sistemas, reduzindo os recursos aos essenciais. NOWEB é independente da linguagem, assim como nuweb [Briggs], LEO [Ream], e FunnelWeb [Ross], que usam uma linguagem de marcação simplificada. Tal abordagem, entretanto, implica numa documentação bem mais pobre. Como o código-fonte não é analisado sintaticamente, é impossível gerar os índices e as tabelas de referência cruzada, que são um dos maiores benefícios da geração automática da documentação.

Os sistemas de programação letrada nunca chegaram a ser amplamente utilizados fora do ambiente acadêmico, de onde provêm os poucos exemplos publicados. São várias as prováveis causas:

- **Dificuldade de integração.** O fonte letrado é trabalhado com editores de texto comuns, usados para edição de programas; isso pode envolver a adaptação dos editores com macros e outros recursos para auxílio à nova sintaxe. O uso de formatos particulares dificulta a integração com ferramentas desenvolvidas por terceiros, que nem sempre possuem um alto grau de adaptabilidade.
- **Visualização tardia.** É necessário tecer/enlear o fonte para verificar como ficará o resultado final; o desenvolvimento de editores WYSIWYG é custoso e tem de ser feito sob medida para cada ferramenta e linguagem de marcação.
- **Baixa escalabilidade.** Alguns destes sistemas exigem que o código marcado seja armazenado num único arquivo. Além de dificultar sua utilização para sistemas de médio e grande porte, isto inviabiliza o trabalho em equipe.

Estes problemas não passaram desapercebidos, e algum progresso foi feito no sentido de prover um ambiente de desenvolvimento integrado voltado especialmente para a programação letrada [Brown-Childs][Brown-Cordes].

O maior entrave à difusão destes sistemas, entretanto, parece ser de caráter filosófico. Tais sistemas pressupõem que o programador tem, antes mesmo de iniciar a escrita do programa, uma idéia clara da arquitetura do sistema e de como organizar o documento de forma clara e didática. Ora, isso só é possível se o programador é um profissional altamente competente (como o Prof. Knuth), e mesmo assim, se o sistema é de pequeno porte e possui arquitetura plenamente definida de antemão. O contraste com a realidade é cruel. A maioria dos profissionais encarregados de redigir uma documentação de qualidade não tem experiência didática, e o seu talento literário pode variar de bom a sofrível. A arquitetura dos programas é muitas vezes radicalmente alterada no decorrer do seu desenvolvimento, e o trabalho de documentar o código é por isso postergado até que se estabilize. Acima de tudo, o desenvolvimento de ferramentas robustas é impulsionado quase que exclusivamente pela demanda de sistemas de médio e grande porte, onde os ganhos de produtividade induzidos pelas ferramentas são palpáveis e o investimento justificado. Do ponto de vista mercadológico, uma metodologia que exija profissionais altamente capacitados, processo de desenvolvimento maduro e altos investimentos em ferramentas é uma metodologia com pouca ou nenhuma chance de sobreviver. Não está em questão a sua qualidade ou valor intrínseco — neste aspecto a programação letrada é uma obra de arte tanto na concepção quanto na implementação — mas sua aplicabilidade é reduzida num universo com programadores não tão brilhantes, *software houses* imaturas e orçamentos limitados.

2.6. Legado

A busca por uma solução mais viável só tem a ganhar analisando os sistemas existentes. As verdades e conceitos neles cristalizados são fundamentais para o sucesso de qualquer solução envolvendo código e documentação:

- Código e comentários estão intimamente ligados, e as sintaxes das linguagens de programação, na forma atual, não expressam esse vínculo apropriadamente.
- Código e comentários pertencem a níveis semânticos diferentes, e qualquer esquema ou gramática que tente mantê-los juntos será intrinsecamente complexo e limitado.
- Estilo e formato são importantes tanto para o código-fonte quanto para a documentação. A legibilidade deve ser otimizada em ambos os níveis.
- A organização hierárquica ideal da informação na documentação raramente coincide com a imposta pela sintaxe da linguagem de programação.
- A retenção de informação sintática e semântica do fonte é imprescindível não só para a formatação como para a geração de documentação útil. Em outras palavras, um esquema completamente independente da linguagem de programação gera uma documentação necessariamente pobre.
- A adoção de padrões internacionais é altamente benéfico em todos os níveis, pois promove a interoperabilidade e reúso das ferramentas.

```
<?xml version="1.0" encoding="UTF-8"?>
<program xml:space="preserve"
         xml:lang="x-C">
int main(void) {
   /* Será isto um programa? */
   return 0;
   }
</program>
```

3. PADRÕES

Como vimos na seção anterior, a maioria dos sistemas de programação letrada e extratores de documentação prevêem o uso de uma linguagem de marcação para explicitar os elementos relevantes do texto. Vamos, nesta seção, rever o que é uma linguagem de marcação e analisar brevemente os padrões e tecnologias afins, ressaltando sua aplicabilidade ao problema de documentação do código.

3.1. Conteúdo e marcação genérica

Num documento eletrônico, marcadores são os códigos, embutidos no texto, que armazenam a informação necessária para o processamento mecanizado, como nomes de fontes, formas de realce ou a estrutura do documento. Todo padrão de documentação eletrônica usa algum tipo de marcação.

O termo marcação originou-se na indústria de publicação tradicional, onde denomina as indicações manuscritas das especificações tipográficas necessárias à

composição e diagramação do texto. A marcação é uma atividade à parte, que acontece entre a redação e a linotipia.

De forma similar, o processamento de texto usa marcações para que o usuário possa especificar sua aparência, como o uso de uma fonte ou de negrito. De um modo geral, o usuário analisa a estrutura do texto e embute comandos de um conjunto predefinido que dirão à impressora como formatar o texto indicado. É interessante notar que a estrutura do texto (títulos, parágrafos, citações, etc.) é o ponto de partida para a aplicação das instruções de formatação — o usuário faz isso inconscientemente. Este tipo de marcação é às vezes denominada marcação procedimental, porque a marcação ativa algum procedimento programado no dispositivo de saída. Há um forte paralelo com a marcação tradicional; a principal diferença é que as marcações são armazenadas eletronicamente.

O formato RTF (*Rich Text Format*), desenvolvido pela Microsoft mas suportado em vários processadores de texto, é um padrão de marcação procedimental. A listagem a seguir mostra um trecho de um documento no formato RTF.

```
{\rtf1\ansi\ansicpg1252\deff0\deflang1033\deflangfe1033
{\fonttbl{\f0\froman\fprq2\fcharset0 Garamond;}
{\f1\fscript\fprq2\fcharset0 Lucida Handwriting;}}
{\colortbl ;\red0\green0\blue255;}
\uc1\pard\sb100\sa100\nowidctlpar\lang3081\ulnone\b\f0\fs36
Exemplo de RTF\b0\fs24\par
\f0 Isto é um texto \iqualquer\i0\f1\par
}
```

Esta abordagem tem vários problemas. Primeiro, ela não registra a estrutura do documento. O usuário codifica a aparência com base na estrutura, mas somente aquela é armazenada, e a informação estrutural é perdida. Além disso, qualquer mudança nas regras de formatação implica em alterar o documento, e a marcação é algo dependente do sistema, reduzindo a portabilidade. Acima de tudo, o esquema torna quase impossível para um ser humano editar diretamente o arquivo, dada a complexidade da marcação.

A marcação acabou evoluindo para uma forma genérica pela introdução de macros, que substituem os comandos com chamadas a funções de formatação externas. Um identificador de estilo ou rótulo é associado a cada elemento de texto e as regras de formatação são por sua vez associadas aos rótulos. Um formatador processa o texto e produz um documento no formato final requerido pelo dispositivo de saída. Um bom exemplo deste tipo de marcação é TEX, o sistema de tipografia criado por [Knuth-91]. A listagem abaixo mostra como se parece um texto em TEX.

```
% exemplo.tex
\nopagenumbers

\noindent Exemplo de documento TEX\par
\smallskip

Isto é um texto comum.\par
\bye
```

Este esquema genérico tem grandes vantagens. Para trocar a aparência do documento, basta adaptar a macro associada, e a mudança é automaticamente propagada por todo o documento; não é preciso reeditar a marcação, que é um procedimento trabalhoso e propenso a erros. O resultado é um aumento de portabilidade e flexibilidade.

Este tipo de marcação está mais perto de capturar a estrutura do texto. Os usuários tendem a dar nomes significativos aos rótulos, refletindo a estrutura lógica do documento e habilitando manipular os trechos automaticamente. Por exemplo, se cada figura do documento é rotulada como `\picture`, é possível gerar mecanicamente um índice de figuras durante a formatação.

3.1.1. SGML

No final da década de sessenta, a GML (*Generalized Markup Language*) foi criada na IBM por pesquisadores chamados *coincidentemente* **G**oldfarb, **M**osher e **L**orie.

Quando Charles Goldfarb assumiu o Comitê de Linguagens de Programação para o Processamento de Texto do ANSI (*American National Standards Institute*), em 1978, a GML já havia se tornado importante no setor de publicações. O primeiro rascunho da SGML (*Standard Generalized Markup Language*) surgiu em 1980, e em 1983 o sexto rascunho já era adotado por várias instituições americanas, em destaque o *Internal Revenue Service* (equivalente à nossa Receita Federal) e o Departamento da Defesa, que passou a demandar que os seus maiores contratados também usassem a SGML. Em 1986, o comitê se expandiu num grupo de comitês colaborando para gerar padrões tanto para o ANSI como para a ISO (Organização Internacional de Padrões). Em 1986, depois de oito anos de trabalho dos comitês, a SGML tornou-se a norma ISO 8879:1986.

SGML é similar à marcação genérica, mas com duas características adicionais importantes:

- a marcação descreve a estrutura do documento, não a sua aparência;
- a marcação respeita um modelo formal, o que significa que o documento pode ser processado por *software* ou armazenado num banco de dados.

SGML não é uma estrutura padrão que todo documento deve seguir, mas uma norma que rege como confeccionar tais modelos de forma padronizada. Isso é mais do que razoável, visto que é irreal achar que um determinado modelo possa se aplicar a qualquer tipo de documento. Os requisitos funcionais de diferentes documentos como relatórios, dicionários, livros, cronogramas, etc. nunca poderiam ser atendidos por um único modelo sem comprometer seriamente o desempenho e a mantenabilidade das ferramentas e do próprio modelo. Ou seja, SGML não é propriamente um modelo de documento, mas uma metanorma que especifica como construí-los de maneira padronizada.

O modelo de um documento SGML é descrito por um conjunto de regras chamado de DTD (*Document Type Definition*), que em geral constituem um documento à parte, referenciado pelo documento principal. Um documento que seguisse um modelo hipotético contido num arquivo *note.dtd* teria o seguinte aspecto:

```
<!DOCTYPE note SYSTEM "note.dtd">
<note>
   <title>Exemplo de documento SGML
   <body>
      <para>Este é um parágrafo exemplar.
      <para>Este é mais um.
</note>
```

Apesar da SGML não impor um determinado modelo a um documento, comitês, corporações e outros grupos normativos utilizam a SGML para criar suas próprias estruturas padronizadas de documento. Um conjunto de elementos definidos por um modelo que se conforme à SGML recebe, na norma, a denominação de **aplicação** SGML. Evitamos usar este termo para não criar confusão com a noção de aplicativo (uma peça de *software*). Vamos citar de soslaio alguns modelos SGML importantes.

3.1.1.1. DocBook

DocBook é um grupo de elementos para descrever livros, artigos e outros documentos em prosa, particularmente documentação técnica. Uma das primeiras aplicações SGML de vulto, começou em 1991 como um projeto conjunto entre a HaL Computer Systems e O'Reilly Publicações. Sua popularidade cresceu e em meados de 1998 sua manutenção passou a um comitê técnico da OASIS - *Organization for the Advancement of Structured Information Standards*. Uma ótima descrição deste modelo, acompanhado de ferramentas, pode ser encontrado em [Walsh]

3.1.1.2. CALS MIL-M-28001B

CALS (*Continuous Acquisition and Life-cycle Support*) é uma iniciativa do Departamento de Defesa estadunidense para promover o intercâmbio de documentos eletrônicos. O padrão MIL-M-28001B especifica modelos para manuais técnicos no formato esperado pelo Departamento, e seu uso foi difundido por força dos contratos milionários onde seu uso é imposto.

3.1.1.3. HTML

Esta é, sem sombra de dúvidas, a mais popular das aplicações SGML. A história da *Hypertext Markup Language* (linguagem de marcação para hipertextos) começa em 1989, quando Tim Berners-Lee e Robert Caillau colaboraram na criação de um sistema de informações ligadas que fosse acessível pelos diferentes sistemas de computadores instalados no CERN (*Conseil Européen pour la Recherche Nucléaire*), em Genebra, na Suíça. Naquela época muitos usavam TEX e postscript na preparação de documentos, e alguns poucos SGML. Tim concluiu que era necessário algo muito mais simples, que pudesse ser usado em qualquer tipo de máquina, desde terminais "burros" operando em modo texto até estações gráficas X-Windows de alto desempenho. A HTML foi então desenvolvida como um conjunto muito simples de elementos, em par com o protocolo de rede HTTP (*Hypertext Transfer Protocol*).

Originalmente, HTML continha principalmente elementos de marcação estrutural (como `<P>` para parágrafos e `<H1>...<H4>` para cabeçalhos) e semântica (como `<TITLE>` para títulos e `<CITE>` para citações). Com a explosão do uso da Internet, mais precisamente da World Wide Web, os fabricantes dos navegadores mais populares, entravados numa guerra comercial, começaram a pressionar revisões sucessivas no padrão para incluir mecanismos de formatação do texto que proporcionassem o mesmo nível de controle que o usuário tinha nos editores WYSIWYG de

seus computadores. O resultado foi uma enxurrada de elementos voltados exclusivamente para a formatação, como `<B>` para negrito ou `<FONT>` para selecionar uma fonte. Mais recentemente, criaram-se até elementos para incluir *scripts* ou mesmo programas inteiros numa página de hipertexto. Tais adições possibilitaram o alto grau de sofisticação gráfica e funcionalidade que vemos nos sítios atuais, e certamente viabilizaram o uso intensivo da Rede do ponto de vista comercial, porém minaram a alta portabilidade da linguagem e perverteram seu espírito original, herdado da SGML, onde conteúdo, formatação e comportamento devem ser mantidos separados para maximizar a mantenabilidade.

Mais recentemente, já sob a tutela do W3C (*World Wide Web Consortium*), o comitê gestor do padrão tem evitado introduzir mais recursos diretamente na linguagem, em favor de outros mecanismos de formatação, como folhas de estilo (que examinaremos na seção 3.2). A versão atual do padrão (4.01) será muito provavelmente a última, já que a linguagem foi reformulada sob a ótica do padrão XML, dando origem à XHTML 1.0 (v. seção 3.1.2.1).

3.1.2. XML

Em meados da década de 90, algumas aplicações SGML já se difundiam em alguns nichos, em especial no ramos de publicações eletrônicas. Sua complexidade, contudo, impedia uma maior disseminação. O Comitê Editorial Revisor da SGML, sob o teto do W3C, achou por bem em 1996 constituir um grupo de trabalho e em fins de 1998 a "Recomendação W3C XML 1.0" [XML] era ratificada . A XML *(Extensible Markup Language)* é descrita na própria recomendação como "um dialeto [ou subconjunto] da SGML" cuja meta " é permitir que SGML genérica seja provida, recebi-

da e processada na Web nos moldes do que é hoje possível com HTML[5]". Por essa razão, "XML foi projetada visando facilidade de implementação e interoperabilidade tanto com SGML quanto com HTML". Todo documento XML válido é, por definição, um documento SGML válido, de forma que as ferramentas usadas para processar SGML não se perdem.

Pode-se ver XML como uma simplificação drástica da SGML, onde procurou-se manter toda a funcionalidade considerada essencial (em particular a generalidade do modelo) e removeu-se uma série de recursos que são pouco usados e de difícil implementação. Por exemplo, em SGML a definição de cada elemento especifica se os marcadores finais e iniciais são obrigatórios ou se um deles pode ser omitido. Por exemplo, o elemento `<P>`, que indica um parágrafo em HTML, prescinde do marcador de fechamento `</P>`: o processador deve subentender que o início de um parágrafo (ou outro elemento que defina um bloco de texto) define automaticamente o término do anterior. Em XML, entretanto, este recurso foi intencionalmente removido: todos os elementos devem abrir e fechar. O mesmo elemento parágrafo em XML deve ser escrito `<P>...</P>`. Elementos desprovidos de conteúdo, como `<BR>`, que sinaliza uma quebra de linha, devem ser escritos `<BR></BR>` ou abreviados como `<BR/>`. Outro exemplo é o caráter de escape usado para representar caracteres especiais: em SGML ele pode ser redefinido à vontade; em XML, é sempre o ampersande (&). Estas diferenças aparentemente pequenas desobrigam os analisadores sintáticos XML de realizar uma série de procedimentos, tornando-os consideravelmente mais simples e rápidos que seus correspondentes SGML. A linguagem foi projetada para tornar o processamento no lado do cliente o mais rápido possível,

[5] O documento se refere às versões 3.2 e 4.0 da HTML, em uso quando da publicação da recomendação, em 10 de fevereiro de 1998.

dentro dos limites de seu objetivo primário como formato para publicação eletrônica e intercâmbio de dados.

Novamente por um imperativo mercadológico, de forma a habilitar sua aplicação em múltiplas áreas, XML é completamente internacionalizada, com suporte a uma pletora de línguas ocidentais e orientais. Todos os processadores XML são obrigados a suportar o conjunto de caracteres universal Unicode nas codificações UTF-8 e UTF-16.

A filosofia de projeto da XML deu resultados imediatos. Sua relativa simplicidade, comparada à SGML, provocou uma avalanche de ferramentas e aplicativos tanto de caráter acadêmico como comercial. Com o apoio de grandes empresas do ramo de *software*, que compreenderam o impacto de uma linguagem unificada para troca de dados, a XML tornou-se o *"hit"* do momento, com todos os benefícios e mazelas que a fama proporciona.

Já por ocasião da publicação da primeira versão da especificação, outros grupos de trabalho no W3C estavam envolvidos no desenvolvimento de vários outros padrões afins. O grupo *XML Namespaces* especificava detalhes do mecanismo que permite remover conflitos de nomes; o grupo XSL projetava uma linguagem específica para transformar e formatar documentos XML; o padrão CSS *(Cascading Style Sheets)*, criado para a HTML, foi adaptado para suportar também XML; o grupo XLink provê um padrão para definir *hyperlinks* sofisticados. A explosão no número de padrões relacionados com XML tem sido apontada por muitos como um problema sério para sua disseminação, mas é apenas um reflexo da rápida assimilação da especificação, fruto de sua simplicidade e generalidade.

O número de aplicações XML cresce dia a dia, incluindo desde linguagens para a definição da estrutura química de moléculas até protocolos para transações bancárias. O setor financeiro, que sofre duma babel crônica causada pela multidão de

leis e sistemas monetários, tem investido muito na tecnologia. Vamos citar abaixo algumas aplicações de utilidade prática para o desenvolvedor de *software*.

3.1.2.1. XHTML

A definição da XHTML *(Extensible Hypertext Markup Language)* começou foi disparada antes mesmo da especificação XML estar pronta, e não traz muitas novidades do ponto de vista funcional: XHTML 1.0 é uma reescrita do padrão HTML 4.01 num modelo XML. Entretanto, a interrupção no desenvolvimento da HTML, cuja quarta versão foi decretada como final, em favor de uma substituta conforme à XML, representa um retomada dos princípios básicos que guiaram a criação da linguagem, com favorecimento da marcação de conteúdo, com a formatação implementada separadamente através de folhas de estilo (agora disponível via CSS ou XSL).

É fato que os fabricantes têm se mostrado extremamente lentos na incorporação dos novos padrões: a maioria nem suporta todos os recursos da HTML 3.2, e só agora, quando já se discute a terceira versão de CSS, estão surgindo os primeiros navegadores com suporte completo à primeira. Contudo, considerando que o suporte a XML está sendo levado a sério pelos fabricantes (o Netscape Navigator armazena toda sua configuração em arquivos XML, e o Microsoft Internet Explorer já incorpora um transformador XSLT 1.0) é de se esperar que a migração para XHTML não tarde, até porque a tradução de HTML 4.0 em XHTML 1.0 pode ser feita com relativa facilidade. Já é comum em várias organizações a obrigatoriedade do uso de XHTML em novos projetos.

3.1.2.2. MathML

Sendo os próprios engenheiros e cientistas da computação os principais usuários das linguagens de marcação, é compreensível que uma das primeiras demandas envolvesse a capacidade de inserir no texto marcado fórmulas e expressões matemá-

ticas. Apesar do conjunto de caracteres universal usado pela XML conter dúzias de símbolos matemáticos, qualquer fórmula que não a mais banal utiliza recursos de formatação para infundir semântica. Com o desenvolvimento da ciência, o número de diferentes notações não pára de aumentar, e seu uso em textos científicos é essencial.

Os que refutam a separação conteúdo-formato que fundamenta as linguagens de marcação costumam citar a notação matemática, junto com a poesia concreta, como exemplos de texto onde a formatação não pode ser dissociada do texto. A caligrafia, sobretudo nas línguas que utilizam caracteres ideográficos, é uma arte per se. Decerto que há casos onde a distinção entre formato e conteúdo é difícil e contra-natural: o formato constitui a própria informação, e as linguagens de marcação não foram criadas para capturar informação visual, mas textual. Não se espera que sejam usadas para capturar uma pintura ou um filme cinematográfico, para tal existem formatos mais apropriados.

O caso das fórmulas é peculiar: a conexão entre a notação matemática e a idéia por trás delas é sutil e profunda. A lógica formal levanta questões ainda pendentes sobre as relações entre os sistemas simbólicos e os fenômenos que eles modelam. Matemáticos e professores entendem bem a importância de escolher uma notação que enfatize os aspectos relevantes de um problema e minimize os de pouco interesse imediato. Em contrapartida, a notação matemática é capaz de um rigor prodigioso, e usada adequadamente, pode descrever objetos de forma inequívoca, o que é importantíssimo para a troca de informações entre sistemas computacionais, como manipuladores algébricos ou sintetizadores de voz (formatação aural).

A MathML *(Mathematical Markup Language)* procura abordar ambas as facetas. Ela define marcadores capazes de representar tanto as notações quanto estrutu-

ras matemáticas. Por exemplo, a expressão $(a+b)^2$ pode ser representada usando marcação de apresentação como:

```
<msup>
  <mfenced>
    <mrow>
      <mi>a</mi>
      <mo>+</mo>
      <mi>b</mi>
    </mrow>
  </mfenced>
  <mn>2</mn>
</msup>
```

onde podemos observar os marcadores `mi`, `mo` e `mn` usados respectivamente para identificadores, operadores e números. A marcação de conteúdo para o mesmo exemplo é:

```
<apply>
  <power/>
  <apply>
    <plus/>
    <ci>a</ci>
    <ci>b</ci>
  </apply>
  <cn>2</cn>
</apply>
```

onde vemos os marcadores `power` e `plus` usados para denotar as operações de exponenciação e adição. Estes dois tipos de marcação podem ser combinados se necessário.

Uma característica da MathML, que é compartilhada por alguns outros dialetos XML, é que ela não define um tipo de documento, mas um conjunto de marcadores que pode ser usado para expressar objetos matemáticos dentro de outros documentos. Um exemplo prático pode ser observado no navegador Amaya, que é capaz de visualizar documentos XHTML com fragmentos de MathML e SVG embutidos.

MathML pode servir ao desenvolvedor tanto na documentação, pela sua simples inclusão no texto dos comentários, como na verificação e geração de código a partir de marcação de conteúdo.

3.1.2.3. SVG

SVG *(Scalable Vector Graphics)* é uma linguagem para descrever gráficos bidimensionais em XML. Ela suporta três tipo de objetos: formas gráficas vetoriais (como por exemplo seqüências de linhas retas e curvas), imagens e texto. Tais objetos gráficos podem ser agrupados, estilizados, transformados e compostos com outros objetos previamente definidos. O texto pode se encontrar em qualquer espaço nominal conveniente à aplicação, o que torna os gráficos em SVG facilmente acessíveis e reutilizáveis.

Os recursos gráficos previstos incluem transformações aninháveis, guias de corte, máscaras-alfa, efeitos de filtragem, objetos de gabarito e extensibilidade. Os gráficos SVG podem ser dinâmicos e interativos, e animação pode ser conduzida através de *scripting* pela manipulação dos elementos XML. Um rico conjunto de manipuladores de eventos (e.g. *onmouseover*, *onclick*) podem ser atribuídos aos objetos gráficos. Apesar de muitos desses recursos já estarem disponíveis em outros formatos, a compatibilidade com outros padrões da Web promovida pelo uso consistente da XML e seus espaços nominais dá-lhe um nível de interoperabilidade sem precedentes. O navegador Amaya, por exemplo, é capaz de entremear livremente gráficos SVG e fórmulas em MathML nos arquivos XHTML.

A Figura 1 mostra o gráfico gerado pelo código SVG na Listagem 1.

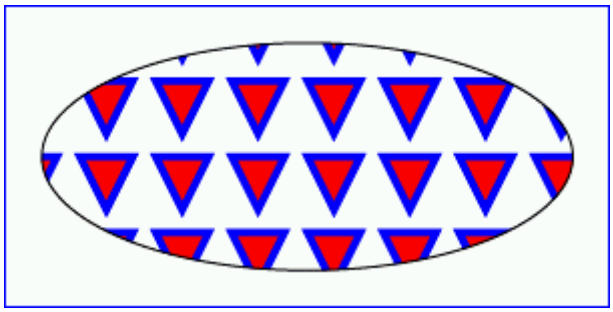


Figura 1 – Um gráfico SVG

```
<svg width="8cm" height="4cm">
  <defs>
    <pattern id="TrianglePattern" x="0" y="0" width="1cm"
             height="1cm" viewBox="0 0 10 10">
      <path d="M 0 0 L 7 0 L 3.5 7 z"
            style="fill:red; stroke:blue"/>
    </pattern>
  </defs>
  <rect style="fill:none; stroke:blue" x=".01cm" y=".01cm"
        width="7.98cm" height="3.98cm"/>
  <ellipse style="fill:url(#TrianglePattern); stroke:black"
           cx="4cm" cy="2cm" rx="3.5cm" ry="1.5cm"/>
</svg>
```

Listagem 1 – Um fragmento de código SVG

O uso de SVG em conjunto com XHTML permite-nos não só incluir gráficos na documentação dos programas, mas vinculá-los diretamente aos elementos correspondentes do código. Diagramas UML associados e trechos de código, como definição de classes, podem ser gerados diretamente por transformações no código e, se tais transformações forem escritas cuidadosamente de forma a não perder conteúdo, o caminho inverso pode ser trilhado com a mesma facilidade.

3.1.2.4. DocBook

A OASIS, responsável pelo padrão DocBook, criou uma versão equivalente em XML que tem ganhado adeptos, pela maior disponibilidade e menor custo das ferramentas. O uso deste padrão para a confecção dos manuais pode otimizar o uso de recursos. Por exemplo, as ferramentas de tecedura, que geram documentação a

partir de código letrado, podem gerar documentos em DocBook, isentando-se de detalhes de formatação e garantindo consistência visual com outros manuais confeccionados diretamente em DocBook por edição manual.

3.1.3. Esquemas

A recomendação XML distingue dois níveis de validação para os documentos. O primeiro define apenas os aspectos de codificação física e exige que o documento possua um e apenas um elemento raiz, e seu conteúdo seja estruturalmente consistente, sem superposições. Isso equivale a dizer que todo elemento deve ou ser vazio ou possuir tanto um marcador inicial e final, e que a ordem de fechamento dos elementos é a ordem inversa de abertura. Por exemplo, a seqüência

```
este é <b>um <i>exemplo</b> de texto </i> problemático
```

é aceita pela maioria dos navegadores, mas não é um fragmento XML válido, pois o elemento `<b>` não poderia ser terminado antes do `<i>`. Neste nível de validação, não importa quais os elementos presentes, apenas que eles formem uma estrutura hierárquica consistente. Um documento que respeite tais condições básicas da sintaxe é dito *bem formado*.

A exigência de o documento ser bem formado é suficiente para o transporte dos documentos XML e para a maioria dos processadores de documentos, como navegadores. Para que o documento possa ser corretamente interpretado em função de alguma semântica, entretanto, é conveniente introduzir um conjunto de restrições adicionais que definem quais os elementos válidos e o que cada um pode conter. Por exemplo, em XHTML, não há imposições quanto ao aninhamento dos cabeçalhos `<h1>`…`<h4>`, mas em DocBook é inválido iniciar um cabeçalho que não seja do nível subseqüente ao envolvente. Isso é bem mais razoável, pois tais elementos definem subseções e iniciar uma subseção de nível 4 dentro do nível 2 não faz muito sentido.

Com muito mais razão, não deveria ser possível inserir elementos `<h1>` em elementos `<h4>`. Este conjunto de restrições adicionais é que na verdade distingue os diferentes tipos de documento XML. Para que o documento possa ser validado corretamente pelo analisador sintático, o conjunto de regras de formação deve estar disponível, seja embutido no próprio documento ou lido de um outro arquivo através de uma referência no cabeçalho.

3.1.3.1. DTD

A DTD *(Document Type Definition)* é o formato básico para as regras de composição dos arquivos XML, e é uma versão simplificada das DTDs usadas para SGML. Este formato possui a vantagem de ser simples e permite o tratamento correto dos documentos XML por ferramentas SGML. Mas tem problemas conhecidos:

- Usa uma sintaxe diferente da usada no documento XML. Se usasse a mesma sintaxe, poderia ser manipulado pelas mesmas ferramentas usadas para o documento principal.
- Este formato é limitado no que tange à definição do conteúdo dos elementos. Por exemplo, `(A B C)` especifica que o elemento deve conter uma instância de `A`, seguida de uma de `B` e outra de `C`, nesta ordem. `(A?(B|C)*)` requer um `A` opcional seguido por um número indefinido de `B`s ou `C`s em qualquer ordem. Mas para especificar que o conjunto deve conter no máximo um de cada, em qualquer ordem, seria necessário escrever `((A?((B? C?)|(C? B?)))|(B?((A? C?)|(C? A?)))|(C?((A? B?)|(B? A?))))`, o que é bem pouco legível e impraticável para um número grande de elementos.
- Não há como precisar o tipo do conteúdo — ele é sempre do tipo texto (cadeia de caracteres). Não há como especificar que o conteúdo atenda a restrições de formato numérico, faixa de validade, etc.

3.1.3.2. XML-Schema

Com vias a sanar os problemas das DTDs, vários outros esquemas foram sugeridos, que culminaram na recente especificação conhecida por XML-Schema. Ela usa a sintaxe XML para definir restrições adicionais de formato e estrutura. Sua aprovação tem sido duramente criticada, principalmente pela falta de um modelo teórico consistente que embase o formato e pelo fato dele ainda conter limitações intrínsecas no tipo de restrições permitidas. Como o padrão ainda não estava consolidado no início de 2001, não foi utilizado neste trabalho.

3.1.3.3. Schematron

O Schematron é um sistema lingüístico criado por [Jellife] para especificar e declarar assertivas sobre padrões arbitrários em documentos XML, baseado na presença, nomes e valores de elementos e atributos ao longo de um caminho. Os esquemas tradicionais são baseados em gramáticas livres de contexto — as regras de derivação permitem especificar a relação somente entre um elemento-pai e seus filhos. A abordagem de reconhecimento de padrões usada pelo Schematron admite regras envolvendo quaisquer elementos, e permite representar muitos tipos de estruturas cujas representações gramaticais são inconvenientes ou difíceis. Em compensação, construções que envolvem repetições são mais difíceis de especificar.

A linguagem é muito simples e ao mesmo tempo genérica, e se aplica especialmente à engenharia de *software*, na representação de requisitos e restrições para modelos de documentos XML, e para pré- ou pós-validar documentos que traficam num sistema. Schematron é implementado sobre XSLT (ver item 3.2.3), utilizando o mesmo mecanismo de casamento de padrões das expressões XPath (item 3.3.1); sua eficiência depende diretamente do transformador XSLT utilizado.

Por uma questão de simplicidade, os exemplos de validação da seção 6 usam diretamente XSLT, mas para implementar conjuntos sofisticados de teste o uso do Schematron é fortemente recomendado.

3.1.4. Metadados

A Internet oferece acesso sem precedentes à informação distribuída ao redor do globo. O uso de metadados melhora sensivelmente a qualidade e rapidez do processo de busca e recuperação da informação, mas sua troca efetiva entre aplicações requer convenções claras sobre semântica, sintaxe e estrutura dos dados. A semântica dos dados é definida por cada comunidade que utiliza os dados para um uso específico. A sintaxe, i.e., o arranjo sistemático dos elementos visando o processamento automático, facilita o uso e a troca dos dados entre aplicações. A estrutura pode ser encarada como um conjunto de restrições formais à sintaxe para uma representação consistente da semântica.

3.1.4.1. RDF

RDF *(Resource Description Framework)* é um modelo baseado em XML para a troca e processamento de metadados. Ele define convenções que facilitam a modularização dos metadados e a interoperabilidade de diferentes conjuntos de elementos descritivos.

O elemento básico do modelo RDF é um terno: um recurso (denominado o *sujeito*) é vinculado a outro recurso (o *objeto*) através de um arco rotulado com um terceiro recurso (o *predicado*). Dizemos que o *sujeito* tem um *predicado* cujo valor é dado pelo *objeto*. Por exemplo, o terno da Figura 2 pode ser lido como "Jaccoud é o criador de lux.html".

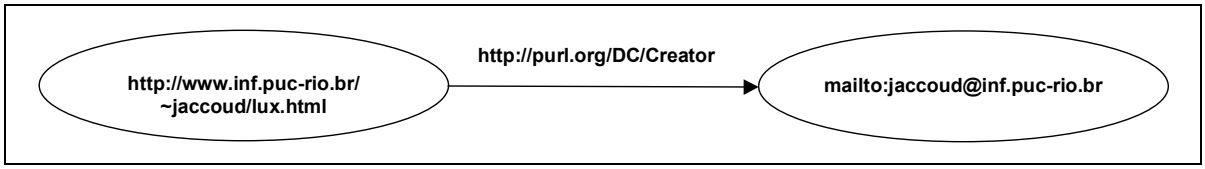


Figura 2 – Um terno RDF

Todos os ternos resultam em um grafo dirigido, cujos nós e arcos são URIs qualificados(6). Um nó pode possuir vários arcos, denotando predicados diferentes. O modelo RDF é muito simples, e acima de tudo, uniforme. O fato de que o vocabulário é composto exclusivamente de URIs permite que um mesmo URI possa ser usado como nó *e* um arco. Isso torna coisas como auto-referência e reificação possíveis, tal como em linguagens naturais.

As metas da RDF são amplas, e as oportunidades potenciais enormes. RDF pode ser usada numa grande variedade de aplicações, por exemplo:

- busca de recursos dirigida;
- catalogação do conteúdo de sítios, páginas e bibliotecas digitais na Internet;
- desenvolvimento de agentes de *software* inteligentes para facilitar a troca de conhecimento;
- descrição de coleções de páginas que compõem um único documento "lógico";
- descrição dos direitos de propriedade;
- definição de preferências de privacidade e níveis de acesso;

(6) Um URI não precisa necessariamente apontar para algo concreto, basta que seja único.

- uso em conjunto com assinaturas digitais, para colaborações em comércio eletrônico.

3.1.4.2. Esquemas RDF

Diferentes vocabulários RDF podem ser usados num mesmo documento, permitindo que as aplicações compartilhem suas definições. Para isso, os vocabulários são concretizados em esquemas, que nada mais são que documentos RDF que descrevem elementos RDF usando um esquema primário (que descreve seus próprios elementos). A mesma uniformidade que caracteriza o padrão se reflete nos esquemas: é possível descrever elementos em função de outros, herdar predicados de outros elementos, etc.

Vários esquemas RDF são mantidos por entidades diferentes. Um dos mais divulgados é o conhecido como Dublin Core [DC], que define vários elementos como *title, creator, publisher, subject, date* e *right.* Cada um deles é minuciosamente descrito usando a norma ISO/IEC 11179 para a descrição de elementos de dado, de forma a promover consistência com outros conjuntos e aumentar a clareza das próprias definições internas. O vocabulário do Dublin Core é um ótimo ponto de partida para a definição de esquemas próprios.

3.2. Forma e estilo: formatação e transformação

3.2.1. CSS

O padrão CSS *(Cascading Style Sheets)* permite que se controle a aparência de páginas em HTML sem que seja necessário editar-lhes o código. Sua característica mais marcante é que os estilos de formatação, como sugere o nome, atuam em cascata: várias folhas de estilo podem ser atribuídas cumulativamente ao hipertexto,

permitindo adaptá-lo aos dispositivos e usuários de maneira progressiva. A especificação define claramente como resolver possíveis conflitos entre as folhas.

A sintaxe duma CSS lembra a da linguagem C. Blocos de propriedades são definidos para diferentes marcadores, indicando propriedades como cor, posição, tamanho e aparência do texto, etc. Os diferentes blocos são aplicados levando-se em conta não só os marcadores, mas seus atributos ou disposição hierárquica, permitindo um controle fino da aparência do texto.

A segunda versão do padrão incluiu adaptações que permitem utilizar folhas CSS também com documentos XML, além de novos recursos para formatação, incluindo a geração automática de texto. Este recurso pode ser usado, por exemplo, para exibir delimitadores ou outros elementos gráficos em função do tipo de elemento e numerar itens automaticamente. Também foi incluído um modelo completo para tabelas.

Apesar de não permitir que os elementos sejam rearranjados, o que limita consideravelmente as opções de formatação, CSS é fácil de aprender e usar, e já está disponível (pelo menos no nível 1) nos navegadores mais recentes. O tipo de formatação requerido para código-fonte, contudo, requer conformidade pelo menos com o nível 2, e são poucos os aplicativos que oferecem este recurso. Uma terceira versão, com capacidade para manipulações simples, já está em elaboração.

3.2.2. DSSSL

A DSSSL *(Document Style Semantics and Specification Language)* é definida pelo padrão ISO/IEC 10179:1996, e enfoca a transformação e formatação, tanto para papel como para meios eletrônicos, de documentos SGML. Com sintaxe e filosofia baseadas em lisp, a linguagem permite aplicar diferentes tipos de formato a vários objetos

durante as fases de composição, leiaute e paginação, e também permite que o usuário especifique transformações de documentos de um tipo de modelo SGML em outro.

DSSSL é complexa, refletindo todo o poderio e generalidade da SGML, o que reprimiu sua disseminação. Apesar da qualidade e versatilidade, são poucos os aplicativos que usam DSSSL.

3.2.3. XSL

XSL *(Extensible Stylesheet Language)* é para XML o que DSSSL é para SGML: uma linguagem que permite especificar transformações e formatação de documentos para uso em sistemas de formatação automatizados. Apresenta todavia algumas características que a tornam singular:

- XSL é, ela mesma, um dialeto XML, e portanto com sintaxe uniforme;
- é otimizada para transformar XML em XML (incluindo XHTML) ou HTML, e portanto com alto potencial para aplicação em servidores de páginas;
- define dois conjuntos bem distintos de elementos, definidos por especificações separadas, um voltado para transformações (XSLT) e outro para a formatação de objetos (XSL-FO); e
- prevê vários mecanismos de extensão.

XSL opera sobre uma API padronizada para acesso à árvore do documento, chamada DOM *(Document Object Model)*, e o fato de ela mesma ser escrita em XML permite utilizar o mesmo processador para o documento-alvo, as transformações e o resultado, otimizando recursos. A primeira parte da linguagem a ser padronizada foi o subconjunto voltado para transformações (XSLT), e em curto espaço de tempo já

surgiram diversos transformadores, uns poucos se distinguindo por um melhor desempenho ou conformidade ao padrão.

O subconjunto dos FOs *(Formatting Objects)* está em fase final de padronização, e já existem alguns projetos implementado interfaces com outros padrões já consagrados de formatação, como o PDF (o formato de documento portátil da Adobe Systems) e TEX.

Uma peculiaridade de XSLT é seu caráter híbrido — a linguagem apresenta características tanto de linguagem funcional como imperativa, mistura casamento de padrões com gabaritos parametrizáveis, tudo isso numa sintaxe pouco convencional. Se o usuário tinha problemas com DSSSL e seu jeito uniforme e lispiano de ser *(**l**ots of **i**rritating **s**uperfluous **p**arenthesis)*, estranhará XSLT pelo motivo oposto: vários paradigmas interagindo para criar um linguagem ao mesmo tempo versátil, esquisita e fascinante. Acresça-se o fato que é possível estender a linguagem tanto através de funções como elementos particulares, e que seu projeto tem restrições para garantir o processamento paralelo dos ramos de transformação, e teremos em mãos uma linguagem com altíssimo potencial, mas cuja assimilação provavelmente será lenta.

Neste trabalho, usamos XSLT integralmente para manipular o hiperfonte, usando TXL apenas para a importação. Ao discutir as transformações, procuramos não nos aprofundar em detalhes de XSLT que podem ser perturbadores, como o tratamento dos caracteres brancos, mas daremos ênfase nos aspectos relacionados à estrutura do documento-fonte e como explorá-la através do uso inteligente de gabaritos e padrões de casamento.

3.3. Referências e vínculos: padrões de associação

3.3.1. XPath

Uma das partes essenciais de XSLT é seu mecanismo de casamento de padrões, que permite caminhar na árvore do documento e selecionar os elementos sobre os quais se deseja operar. Isso é implementado através de um padrão à parte, denominado XPath, e as expressões por ele definidas são compartilhadas em vários ramos da família XML.

XPath define um conjunto de *eixos* na árvore sobre os quais podemos navegar. Os principais são: *ancestor, atribute, child, descendant, following, namespace, parent, preceding* e *self.* Usando palavra-chaves e sinais de pontuação pode-se mover o foco de seleção para pontos-chaves do documento, como a raiz ou nós indexados, e caminhar em qualquer eixo a partir do ponto corrente, por exemplo, no eixo dos atributos de um elemento. Usando predicados após um nome, é possível filtrar os nós selecionados, por exemplo para obter todos os descendentes cujos atributos satisfaçam a uma condição.

Expressões XPath podem também conter chamadas a funções internas ou definidas pelo usuário. As funções pré-definidas podem converter o tipo do resultado de uma expressão, operar aritmética simples, manipular cadeias de caracteres, recuperar informações sobre o processador, o documento, o contexto ou um nó, além de realizar busca por chave ou ID e carregar outros documentos.

3.3.2. XPointer

Uma URI permite endereçar qualquer recurso disponível, mas não permite olhar para dentro dos documentos. O padrão XPointer, ainda em elaboração, estabe-

lece uma forma padronizada para endereçar não só os elementos internos a um documento mas também faixas de conteúdo e outras estruturas internas. Sua sintaxe usa expressões XPath para identificar os limites do trecho endereçado.

Ao tratar os comentários como metainformação, é necessário estabelecer um mecanismo de apontamento para que se indique a que trecho do código o comentário se refere; isto pode ser feito naturalmente com expressões XPointer. Como o padrão ainda não está pronto, refreamo-nos em utilizá-lo nos nossos exemplos, mas já existem exemplos práticos de sua utilização bem sucedida. O navegador Amaya, por exemplo, desenvolvido no âmbito do W3C, implementa um mecanismo de anotações. Uma anotação é um documento RDF externo ao documento XHTML, possivelmente criado por terceiros, e que usa um XPointer para indicar o exato local do documento ao qual se refere.

3.3.3. XLink

O padrão XLink estende o conceito de *link* unidirecional da HTML para incluir *links* bidirecionais e múltiplos, usando se necessário arquivos externos ao documento principal. Sua implementação nos navegadores e ferramentas deve demorar. Entretanto, é possível utilizar a sintaxe XLink (que é um dialeto XML) para representar os vínculos múltiplos de um programa letrado, deixando a interpretação dos *links* a cargo dos processadores encarregados de operar sobre o hiperfonte. (Por exemplo, durante a tecedura, um arquivo com *xlinks* indicando quais rotinas invocam outras pode ser usado para criar um mapa bidirecional de navegação, como o usado por For-See. [Amaral-00]).

```
int main(void) {
   /* C'est ne pas un programme! */
   return 0;
   }
```

4. UM MODELO DE HIPERFONTE

4.1. Código + comentários

Formalmente, um programa é um uma seqüência de instruções codificadas e validadas usando uma gramática. Costuma-se fazer distinção entre programa-objeto, que é o código diretamente executável, escrito na linguagem da máquina, e programa-fonte, escrito numa linguagem de mais alto nível, legível e editável por um ser humano. Às vezes se esquece que o objetivo de usar linguagens de alto nível é tornar os programas legíveis e compreensíveis. A aplicabilidade de uma linguagem é diretamente proporcional à sua capacidade de expressar a solução de uma classe de problemas de forma simples e clara. Para torná-las aplicáveis a problemas cada vez mais complexos, as linguagens são dotadas de mecanismos de abstração que permitam ao programador organizar e isolar suas diferentes facetas com menos esforço. Tal

evolução não é grátis: como a complexidade é transferida para o tradutor[7], e às vezes se reflete na qualidade do código-objeto. Há porém que se tomar muito cuidado: se a forma de estruturar o problema para resolução pelo computador for complexa ou apenas diferente da forma manual, a documentação deve ser aperfeiçoada de forma a compensar. Por mais clara que seja uma linguagem de programação, ela ainda está muito longe da expressividade da linguagem natural.

Assim sendo, é de se espantar que apesar de a legibilidade ser uma das propriedades mais importantes de uma linguagem de programação, a maneira de escrever e comentar o código mantenha-se estagnada. Na década de 1960, os estudos de lingüística computacional ajudaram promover o abandono dos formatos tabulares do texto (fortran, cobol) em função de um formato livre, onde os espaços são considerados delimitadores (pascal, lisp e inúmeras outras linguagens). Mais recentemente, algumas linguagens, como Z e python, levam em conta a disposição dos elementos no texto numa tentativa de melhorar a legibilidade, mas essa característica pode causar problemas se o editor e o compilador não tratarem os caracteres em branco (espaços, tabuladores e quebras-de-linha) de forma consistente e uniforme. Sob o ponto de vista da codificação, isso não é avanço, mas retrocesso, pois o tratamento diferenciado das tabulações e quebras-de-linha é problema conhecido quando a questão é portabilidade.

Quando se trata dos comentários, a situação é pior. Um comentário ainda é apenas uma cadeia de caracteres inserida em meio ao código, da mesma maneira que os cartões perfurados com comentários eram inseridos na pilha. Os comentários

---

[7] A maior parte das alegações neste texto são válidas para compiladores, interpretadores ou outros tradutores de caráter misto. O termo *tradutor* é usado para designar qualquer um dos tipos, e os termos específicos são usados quando se faz necessária a distinção, seja porque a asserção não é válida para todos ou porque não se lhe aplicam.

ainda são considerados como lixo pelo compilador, que trata de extirpá-los do texto logo que possível. Por exemplo, a norma [ISO-C++], na seção 2.1 (fases de tradução), requer que os comentários sejam substituídos por um espaço imediatamente após a conversão dos trígrafos e caracteres estendidos e da emenda das linhas terminadas por uma contrabarra . A ferramenta **javadoc**, que acompanha as distribuições da linguagem java, é uma iniciativa no sentido de resgatar a semântica velada dos comentários. Mas o conjunto de marcadores e as regras de posicionamento não constam da definição da linguagem, que trata os comentários com o descaso costumeiro.

4.2. Código × comentários

A atitude conservadora dos projetistas das linguagens é perfeitamente compreensível. Afinal de contas, ao compilador interessa processar o código rápida e eficientemente, e o esforço adicional de processar os comentários pode ser restritivo: seu volume, em fontes de qualidade, pode chegar ao quádruplo do código propriamente dito. A dilatação do tempo de compilação de um programa quando se gera informação adicional para um depurador ou *profiler* é uma amostra de como esse processamento é dispendioso. Entretanto, seria altamente desejável dotar os compiladores e ligadores de recursos para o processamento de comentários, hoje só disponíveis através de ferramentas distintas. Além de aproveitar todo o analisador léxico e sintático, tal incorporação garantiria uma interpretação consistente do fonte, que é difícil de conseguir na presença de extensões da linguagem ou dependências de plataforma: são poucos os programas que podem se dar ao luxo de ser completamente independentes de plataforma, e mais raros ainda compiladores 100% conformes aos padrões.

Para que os comentários possam ser processados, é necessário incluí-los na gramática da linguagem. Se quisermos que esta nova sintaxe estendida seja perfeitamente equivalente à anterior, estaremos provavelmente assumindo uma tarefa vultuosa, porque, na maioria das linguagens modernas, os comentários podem aparecer em qualquer posição do programa onde um espaço em branco é admitido. Se a associação dos comentários com os demais termos gramaticais se der apenas pela sua posição relativa, seria necessário definir o comportamento para cada caso. Isso tornaria impossível associar comentários a qualquer elemento não terminal, sob pena de ambigüidade. Por exemplo, no trecho em C++

```
int i = 1+2;      // comentário
```

é impossível inferir se o comentário se refere à nova variável i, ao termo iniciador, à soma, a uma das parcelas, ou mesmo à inicialização ou à própria declaração. Para se dar uma idéia da complexidade potencial de uma expressão, observe o porte da árvore sintática gerada pelo interpretador TXL quando aplicamos a gramática C++ do apêndice A à linha acima, aparentemente simples:

```
[program
 [translation_unit
  [repeat declaration [repeat declaration+
    [declaration
     [block_declaration
      [simple_declaration
       [repeat decl_specifier [repeat decl_specifier+
         [decl_specifier [type_specifier [simple_type_specifier ['int']]]]
         [repeat decl_specifier [empty]]
        ]]
       [list init_declarator [list init_declarator+
         [init_declarator
          [declarator
           [direct_declarator
            [declarator_id [id_expression [unqualified_id [id 'i']]]]
           ]]
          [opt initializer [initializer
            ['=']
            [initializer_clause
             [assignment_expression
              [conditional_expression
               [logical_or_expression
                [logical_and_expression
                 [inclusive_or_expression
                  [exclusive_or_expression
                   [and_expression
                    [equality_expression
                     [relational_expression
                      [shift_expression
                       [additive_expression
                        [additive_expression
                         [multiplicative_expression
                          [pm_expression
                           [cast_expression
                            [unary_expression
                             [postfix_expression
                              [primary_expression
                               [literal
                                [integer_literal
                                 [decimal_literal '1']]]]]]]]]]
                        ['+']
                        [multiplicative_expression
                         [pm_expression
                          [cast_expression
                           [unary_expression
                            [postfix_expression
                             [primary_expression
                              [literal
                               [integer_literal
                                [decimal_literal '2']]]]]]]]]]]]]]]]]]]]]]
         ]
         [list_opt_rest_init_declarator [empty]]
        ]]
       [';']
      ]]]
    [repeat declaration [empty]]
   ]]]
]
```

Listagem 2 – Árvove para uma linha C++

São tantos os níveis sintáticos e semânticos superpostos que qualquer mecanismo associativo baseado em posição será extremamente limitado. Sim, é possível utilizar heurísticas para escolher uma associação "sensata", mas a definição de uma linguagem não deve abrir espaço para dúvidas, cada detalhe deve ser definido de forma precisa. Só restam duas opções:

- limita-se drasticamente o posicionamento dos comentários, restringindo sua associação a alguns elementos-chaves, como variáveis e rotinas;
- utilizam-se marcações internas ao comentário para indicar o termo associado do código.

A primeira solução é adotada pelas ferramentas para a geração de documentação, e tem a vantagem de manter a sintaxe original intacta. Um novo conjunto de regras dita as posições onde os comentários devem ser colocados para serem associados automaticamente a um elemento-chave do programa. Mesmo as ferramentas que definem marcadores para classificar o conteúdo dos blocos de comentário, como Documentu e **javadoc**, utilizam sua posição para associá-los corretamente. Algumas aplicações usam regras heurísticas para determinar a associação de alguns itens, mas não é possível controlar sua aplicação ou o resultado, que nem sempre corresponde ao esperado.

A alternativa é indicar explicitamente a qual elemento um comentário se refere. Mesmo um texto narrativo pode ser associado, seja ao elemento subseqüente ou ao continente. Neste caso, a associação não precisa se restringir a elementos terminais, pois é possível referenciar qualquer elemento sintático do programa. Isto é especialmente importante em programas letrados, pois a narrativa via de regra se refere a vários elementos, às vezes distantes no código. Se a sintaxe usada para a associação for suficientemente genérica, é possível indicar não só um elemento

específico, mas também faixas de texto e trechos não contíguos, mesmo que localizados em arquivos diferentes.

O aspecto mais interessante desta última abordagem, cuja generalidade confere-lhe uma extensa vantagem sobre a primeira, é que ela permite dissociar completamente o código dos comentários. Como a associação é indicada explicitamente, não existe necessidade do comentário situar-se próximo ao elemento. Na verdade, nem é preciso que esteja no mesmo arquivo. O termo associativo, que define o vínculo, tanto pode constar do código, do comentário ou de um terceiro elemento, desde que haja uma forma de referenciar os dois primeiros. Codificar um fonte desta forma é, acima de tudo, mais coerente. O código e os comentários, apesar de relacionados, pertencem a níveis semânticos distintos, e possuem requisitos sintáticos muito diferentes. Qualquer tratamento formal que almeje ser genérico e completo deve levar isto em conta. Isso não significa que código e comentários devam usar *mecanismos* de marcação diferentes, apenas que a semântica envolvida é diferente, o que induz *vocabulários* e *esquemas* de marcação diferentes.

Aparentemente, manter código e comentários em arquivos separados traz desvantagens. Pode-se argumentar que é mais fácil manter os comentários atualizados quando eles estão no mesmo arquivo. Ora, certamente não é mais fácil editá-lo. Um arquivo bem comentado pode chegar a 20% de código contra 80% de comentários. Fazer modificações neste código seria bem mais simples se os comentários pudessem ser colapsados e visualizados sob demanda, ou se fossem mostrados em paralelo, e não misturados ao código. Sabemos também que o código raramente é modificado num só ponto, e as alterações costumam proliferar, principalmente quando feitas nas interfaces dos módulos. E que é comum o código ser editado muitas vezes até ser considerado estável, quando então os comentários são atualizados. O que se faz necessário é um mecanismo que verifique a consistência entre o código e as marcações, auxiliando o usuário a manter o conteúdo dos comentários

atualizado, não apenas facilitando sua edição. Manter arquivos separados facilita a criação de tais ferramentas, pois, como vimos, os esquemas de marcação para código e comentários são necessariamente diferentes. Um editor inteligente pode carregar os dois arquivos simultaneamente, de forma que um fonte aparece para o programador como uma só entidade, mesmo que contido em dois arquivos diferentes. Também é possível marcar automaticamente comentários como desatualizados quando o código ao qual se referem for editado, auxiliando seu processo de revisão.

Como veremos a seguir, o ato de segregar os comentários do código e utilizar marcações específicas para os dois arquivos facilita o seu processamento, permitindo até mesmo o uso de ferramentas genéricas.

4.3. Codificando em XML

A idéia de se usar uma linguagem de marcação genérica para armazenar código-fonte não é nova. [Germán] e [Cowan] mostraram como a legibilidade do código pode ser melhorada utilizando SGML e suas folhas de estilo. Mais recentemente, presenciamos o aparecimento de várias ferramentas que usam XML para codificar programas letrados. Os esquemas usados em cada caso variam muito, mas as vantagens proclamadas são comuns: mais *links*, facilidade de formatação e edição inteligente.

4.3.1. *Links*

Usando XML, é possível criar *links* de e para outros documentos com facilidade, o que é de especial utilidade para adicionar documentação que não pode ser incluída diretamente no código. O padrão XLink, em vias de ser efetivado, habilitará,

quando implementado, uma navegação bem mais versátil do que a provida hoje pelos *links* unidirecionais da HTML.

É importante notar que a relação entre os comentários e o código-fonte é que a relação não é biunívoca: um comentário pode referir-se a diversos trechos do programa e um trecho pode ser referenciado múltiplas vezes quando implementa vários aspectos do programa. Esta característica é conscientemente ignorada pela maioria dos modelos, que tentam impor ao texto narrativo uma estrutura hierárquica que não lhes é natural. Ao usar *links*, evita-se que tais comentários sejam repetidos ao longo do código. A forma de implementar esse mecanismo dependerá em grande parte do suporte provido no futuro pelos navegadores e editores.

4.3.2. Formatação

Ao manter-se o conteúdo separado do formato, não se ganha apenas na facilidade de manutenção, mas também em versatilidade, pois o mesmo documento pode ser formatado de várias maneiras, conforme o dispositivo de destino ou o gosto do usuário. Segundo [Bellay], um dos problemas observados com ferramentas de reengenharia é a deficiência de mecanismos de adaptação e extensão mais acessíveis ao usuário final.

Através do uso de diferentes folhas de estilo:

- cada usuário pode visualizar o código no seu estilo preferido (fontes, cores, endentação, etc.);
- é possível formatar o código como texto ou como diagramas;
- diferentes mídias podem ser abordadas independentemente, com a formatação apropriada..

A linguagem de formatação XSL possui aplicações que vão além de prover estilos: transformações complexas podem ser realizadas no código, apesar da ausência duma garantia formal de preservação da semântica.

4.3.3. Edição sensível à sintaxe

A depender do modelo de documento utilizado, um grau considerável de sensibilidade à sintaxe pode ser imbuído nos editores. Apesar das dificuldades inerentes à sua construção, recursos como validação em tempo de edição do código e o acesso sincronizado à documentação de referência são altamente desejáveis. Existem vários editores especializados na edição de código-fonte (em ASCII), e a maioria deles suporta um grande número de linguagens de programação, porém recursos sofisticados como esses só se encontram em uns poucos, especializados nesta ou naquela linguagem. A utilização de um esquema genérico de modelagem do fonte permite implementar tais recursos mais facilmente, usando editores XML genéricos. É impossível desvencilhar-se completamente das peculiaridades inerentes às linguagens de programação, mas as semelhanças podem ser tratadas de maneira unificada, com economia de esforço. Por exemplo, os mecanismos de modularização de C++ (que usa espaços nominais) e java (baseada na correspondência pacote-diretório) são diferentes o suficiente para que o mecanismo de sincronia com a documentação tenha de ser sintonizado para cada linguagem separadamente. Mas a semelhança das duas sintaxes é tal que a maior parte dos mecanismos de apoio à edição (pareamento de colchetes, blocos colapsáveis, etc.) pode ser compartilhada. Alguns aspectos se aplicam a várias linguagens: a inserção automática de um bloco de comandos durante a edição é praticamente idêntica em C ou pascal, diferindo apenas nos delimitadores (chaves ou as palavras-chaves `begin/end`, respectivamente).

Alguns alardeiam como vantagem o fato de que em XML não é necessário usar delimitadores, como parênteses, colchetes ou vírgulas. Isso é uma ilusão: os delimitadores são apenas substituídos por marcadores, inclusive com prejuízo numérico. Por exemplo, a expressão fortran `call func(3,y)`, traduzida segundo o esquema do apêndice A, fica:

```
<call>
  <nm>func</nm>
  <expr><lit t='int'>3</lit></expr>
  <expr><nm>y</nm></expr>
</call>
```

onde os argumentos estão agora delimitados pelos marcadores `<expr>`. O resultado é decerto mais prolixo e menos legível sem uma folha de estilos apropriada; um modelo XML só traz vantagens concretas se acompanhado de ferramentas apropriadas para visualização e manipulação dos documentos. A escolha de XML certamente passa ao largo do quesito concisão, em troca de universalidade e simplicidade. Entretanto, com uma folha de estilos apropriada, o trecho acima pode aparecer como ***call*** **func**(3, **y**), onde os termos `func` e `y` contêm *links* subjacentes que remetem ao código da sub-rotina e à declaração da variável (explícita ou implícita).

Outrossim, o trecho marcado permite que o editor se comporte de forma muito mais inteligente. Vamos imaginar uma seqüência hipotética de entrada da expressão acima num editor que esteja ciente do modelo *for.dtd*. (Detalhes foram omitidos da lógica para simplificar o exemplo.)

- O cursor está parado numa linha em branco, que corresponde a um elemento vazio auxiliar criado pelo editor. O usuário digita 'C'. O editor simplesmente ecoa o caráter 'c' na tela.
- O usuário digita 'A-L-L-espaço'. O espaço ativa uma macro que instancia um elemento `<call>` contendo um elemento `<nm>` para conter o nome obrigatório da rotina. (Na realidade, o editor já poderia agir

após a digitação do ‘a’, mas isso seria prejudicial para a usabilidade do editor. Automação em excesso pode ser desagradável, como podemos observar em alguns sistemas de edição de texto “amigáveis” que ao invés de ajudar, atrapalham a edição. Após tomar um mínimo de intimidade, os usuários invariavelmente acabam desligando quase toda a automação.)

- Uma lista das possíveis sub-rotinas a chamar é mostrada, obtida por uma varredura do arquivo ou de um índice externo.
- Ele digita ‘F-U-N-C’. Os caracteres são selecionando elementos da lista, até que não há mais casamento e ela desaparece. A cadeia “func” é assumida como o nome da sub-rotina.
- O usuário entra um ‘(‘. O editor assume que ele quer acrescentar um argumento (ou índice) e cria um elemento `<expr>` para contê-lo.
- Um ‘3’ é digitado seguido de uma vírgula. O conteúdo é reconhecido como um literal inteiro, encapsulado, e a vírgula dispara a inserção de outro elemento `<expr>` para o segundo argumento.
- Digita-se ‘Y’, seguido de um parêntese ou ‘Return’. O nome ‘y’ é reconhecido como um identificador, envolto num elemento `<nm>` e o editor entende que a entrada do comando terminou, passando imediatamente para uma nova linha em branco.

Um editor real teria lidar com uma lógica muito mais complexa, para levar em conta todas as possibilidades e formas intermediárias. Na seção 7.1, listamos alguns requisitos e sugerimos estratégias para uma implementação especializada na edição de código-fonte. Resta observar com otimismo que mesmo um editor XML genérico é capaz de identificar estruturas obrigatórias na DTD e inseri-las automaticamente no documento, sem que seja necessário nenhuma adaptação.

4.3.4. Ferramentas e trabalhadores

O maior obstáculo à introdução de um novo modelo de codificação para os fontes é de caráter cultural e não técnico: o usuário é avesso a mudanças. O ser humano gosta de trabalhar de forma rotineira porque isso lhe traz a sensação de segurança e autoconfiança. Mudanças geralmente significam a quebra da rotina de trabalho e são sempre acompanhadas de um período de aprendizado e baixa eficiência. Às vezes, a mudança pode significar um prejuízo permanente para determinado grupo, o que só contribui para aumentar a aversão.

Quando a mudança é eficaz, entretanto, em curto espaço de tempo os benefícios se tornam atraentes o suficiente para remover a desconfiança. Quando os terminais interativos foram introduzidos na década de 60, possibilitando escrever o código diretamente no computador ao invés de usar cartões perfuráveis, houve uma resistência muito grande. Os usuários ressentiam-se da ausência de ter às mãos uma representação concreta do programa — a pilha de cartões — e não tinham confiança no meio magnético de armazenamento. Mas a facilidade de edição, principalmente nos grandes programas, rapidamente arrebanhou até os mais renitentes.

A migração do texto comum para um texto marcado deve levantar semelhante maré de receios, principalmente no que tange à falta de ferramentas especializadas. O usuário que, no intuito de disparar uma busca usando `grep`, descobrir que não existe uma ferramenta equivalente, não se mostrará inclinado a abdicar da facilidade sem uma compensação adequada. Compiladores, depuradores, *profilers* e várias outras ferramentas que hoje tratam diretamente com o código, só virão a aceitar código marcado se seu uso for largamente difundido, e é improvável que isso ocorra a curto ou médio prazo. Portanto, uma das características principais de qualquer novo esquema de codificação é que ele possa importar e exportar código em formato comum. Isso permite interagir com as ferramentas hoje existentes de

forma pacífica, até que um determinado modelo se firme. Num futuro ideal, novas versões da linguagens de programação incluiriam nas suas definições um modelo XML equivalente, possibilitando a construção de ferramentas universalmente compatíveis.

4.4. Modelando o código

4.4.1. Opções

Diversos modelos já foram considerados para armazenar código-fonte em XML. [Armstrong] compara os mais comuns, apontando falhas e vantagens de cada. Podemos distinguir três tipos de abordagem e de modelo, que chamaremos de minimalista, plurilíngüe e dedicado.

4.4.1.1. O modelo minimalista

Nesta abordagem, somente alguns poucos elementos primários, como comentários, identificadores, operadores e sinais de pontuação são usados, e permite-se misturá-los à vontade. Alguns modelos mais radicais, como o proposto por [Armstrong], propõem que se reduza tudo a dois tipos de elementos: um para comentários, que pode conter fragmentos de código XHTML, e outro que representa um nó genérico de código, e que pode conter tanto texto como outros nós. Este modelo é muito conveniente para a construção de editores, pois os nós podem ser manipulados facilmente e de maneira uniforme. Todavia a classificação dos nós e sua taxinomia envolve uma análise de casamento de padrões intensa, o que dificulta a construção de qualquer programa de transformação que não seja trivial. Este tipo de modelo é tão pobre que não captura nem mesmo os elementos léxicos mais básicos, como identificadores, constantes literais e pontuação.

4.4.1.2. O modelo plurilíngüe

A idéia de criar um modelo que contemple todas as linguagens de programação, uma "superlinguagem", desvanece ao constatarmos a enorme diversidade de paradigmas e estilos existentes. É possível, contudo, criar um modelo que capture os conceitos mais comuns de uma família de linguagens, como por exemplo as imperativas. Funções, classes, enumerações, e outros elementos são encontrados, salvo pequenas variações, em várias linguagens de forma semelhante.

Um esquema interessante nesta categoria é o [CSF] *(Code Structure Format).* Ele define marcadores para os estruturas básicas de C, C++, common-lisp, python e java. O código marcado é gerado por tradutores e o modelo não foi criado para contemplar edição manual, pois faz uso extensivo de atributos. Um *CSF-linker* é capaz de aglutinar os arquivos de diferentes fontes num único arquivo-mestre sobre o qual trabalham as demais ferramentas, como um extrator de documentação ao estilo **javadoc** e um gerador de diagramas.

Modelos plurilíngües tendem a ser tendenciosos. Por exemplo, CSF usa o termo *package*, oriundo do java, para qualquer conjunto de rotinas, independentemente da linguagem. À medida que o suporte a outras linguagens são adicionadas, tais discrepâncias tendem a aumentar, levando à criação de atributos para indicar a semântica desejada e acabando por degenerar o espírito ecumênico do modelo.

Tais modelos se prestam bem à codificação de sistemas escritos em mais de uma linguagem, o que está se tornando cada vez mais comum em face à multiplicidade de níveis e à distribuição de tarefas entre módulos especializados. A regeneração do código original a partir do marcado é contudo dificultada pela própria característica uniformizadora do modelo: qual a linguagem fonte usada em cada

módulo? A arquitetura SDS *(Software Development Foundation)*, construída sobre o CSF, não prevê a operação de enleio.

O maior benefício de um modelo plurilíngüe é a economia de esforços na construção de ferramentas, às custas de uma limitação drástica da sua especificidade e do tipo de ferramentas possíveis. Não há como realizar transformações genéricas num código sem uma marcação que reflita de perto a sintaxe da linguagem.

4.4.1.3. O modelo dedicado

Nesta abordagem, a sintaxe da linguagem de programação é modificada e adaptada ao modelo XML, procurando reproduzir de perto as estruturas originais. É importante observar que o nível de especificação do dialeto XML oferecido pelas DTDs é pobre, e mesmo com os avanços oferecidos pelo novo padrão de esquemas *(XML-Scheme)* será impossível reproduzir todas as nuances da gramática de forma a proporcionar uma validação automática do texto. Aparentemente vantajosa, esta característica traria contudo muitos problemas na implementação de editores e nos ajustes de variações nas linguagens, necessárias para suportar extensões deste ou daquele compilador. Mesmo as definições da linguagens de programação deixam alguns aspectos da linguagem indefinidos, de forma a dar um certo grau de liberdade aos implementadores, que precisam conviver com características próprias de cada plataforma.

O uso de um esquema para cada linguagem de programação pode parecer restritivo a princípio, mas o mecanismo de espaços nominais da XML permite que criemos documentos compostos, utilizando marcadores de vários vocabulários simultaneamente. Um exemplo disso é o uso de MathML e SVG embarcados em XHTML.

Este tipo de modelo, que foi o adotado neste trabalho, permite um controle preciso de cada aspecto do código-fonte, e é o único que possibilita o desenvolvimento de ferramentas específicas de validação, inviáveis em modelos plurilíngües ou minimalistas. O esforço de se escrever DTDs para cada linguagem não é tão grande quanto parece, porque a especificidade de linguagem nos permite criar modelos conceitualmente mais simples, mesmo que com uma diversidade maior de marcadores. Estabelecidos o nível de profundidade desejado e os elementos sintáticos relevantes, a tarefa de retratar a gramática original numa DTD é razoavelmente mecânica. Como veremos na seção 6.1, se a própria linguagem de descrição da gramática for transposta com sucesso, podemos processar a definição da linguagem para ajudar na implementação das ferramentas.

4.4.2. Marcações

Vamos examinar a seguir algumas aspectos da composição da DTD para uma linguagem de programação. Durante o processo, ressaltaremos as implicações práticas de algumas decisões e extrairemos regras práticas para orientar novas transposições.

Para iniciar o processo, é importante que tenhamos à mão uma gramática formal da linguagem de programação, que servir-nos-á de guia. As normas que definem as linguagens costumam vir acompanhadas de gramáticas, em notações semelhantes à EBNF. Para diminuir a entropia, já que este trabalho trata de várias linguagens, resolvemos adotar a notação utilizada por TXL, uma linguagem para manipulação de fontes em ASCII, da qual trataremos com mais detalhe na seção 5. De uma maneira geral, a produção EBNF

```
E ::= C | E + E | (E)
```

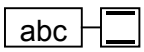
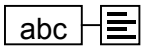
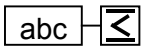
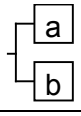
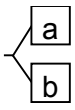
corresponde em TXL a

```
define E
   [C]  |  [E] '+ [E]  |  '( [E] ')
end define
```

Os não-terminais e *tokens* em TXL são sempre delimitados por colchetes, e os literais, em caso de ambigüidade, podem ser precedidos por um apóstrofo. Modificadores são usados para definir implicitamente os tipos derivados mais comuns:

forma extensiva	*forma abreviada*	*cardinalidade*
`[opt A]`	`[A?]`	0…1
`[repeat A]`	`[A*]`	0…∞
`[repeat A+]`	`[A+]`	1…∞
`[list A]`	`[A,]`	0…∞ (separados por vírgulas)
`[list A+]`	`[A,+]`	1…∞ (separados por vírgulas)

Para facilitar a visualização dos fragmentos das DTDs, usaremos ocasionalmente os diagramas gerados pelo aplicativo Near & Far Designer, um editor de DTDs. A simbologia é muito simples e análoga à usada nas DTDs, e permite obervar a estrutura hierárquica, que não é aparente na definição textual.

Símbolo	*Correspondência na DTD*	*Significado*
	`!ELEMENT abc EMPTY`	O elemento `abc` não admite conteúdo.
	`!ELEMENT abc (#PCDATA)`	O elemento pode conter texto.
	`!ELEMENT abc ANY`	O elemento pode conter uma mistura de texto e outros elementos quaisquer, em qualquer ordem.
	`a, b`	Conjunção: um elemento `a` seguido de um elemento `b`, nesta ordem.
	`a | b`	Disjunção: um elemento `a` ou um elemento `b`, mas não ambos.
	`…?` (sufixo)	Opção: zero ou uma ocorrência do elemento.

✳	…* (sufixo)	Repetição: zero ou mais ocorrências do elemento.
+	…* (sufixo)	Repetição: uma ou mais ocorrências do elemento.
%pe	!ENTITY % pe …	Entidade paramétrica (símbolo auxiliar interno à DTD). Usado para facilitar a repetição de expressões complexas.

Adicionalmente, um til (~) aparece à direita do nome do elemento se este pode conter atributos. O diagrama não ajuda a visualizar as definições de atributos, mas, como veremos adiante, estes não cumprem no nosso caso um papel tão importante quanto em outros modelos. Para garantir uma edição consistente, um elemento com conteúdo só aparece expandido em um único ponto do diagrama. Quando é expandido num local, as demais instâncias visíveis aparecem com uma barra vertical espessa à direita, indicando que o conteúdo está visível alhures.

A Figura 3 mostra um diagrama parcial da DTD para o fortran-77. Note que alguns elementos parecem não ter conteúdo — eles estão colapsados. Essa característica permite observarmos cada fragmento da DTD mais facilmente. DTDs simples, como a usada para TXL, possuem diagramas que, mesmo completos, podem ser visualizados numa só página, porém é impossível tratar modelos mais complexos sem colapsar alguns ramos para analisar outros. No apêndice, podemos encontrar o texto e completo desta e doutras DTDs.

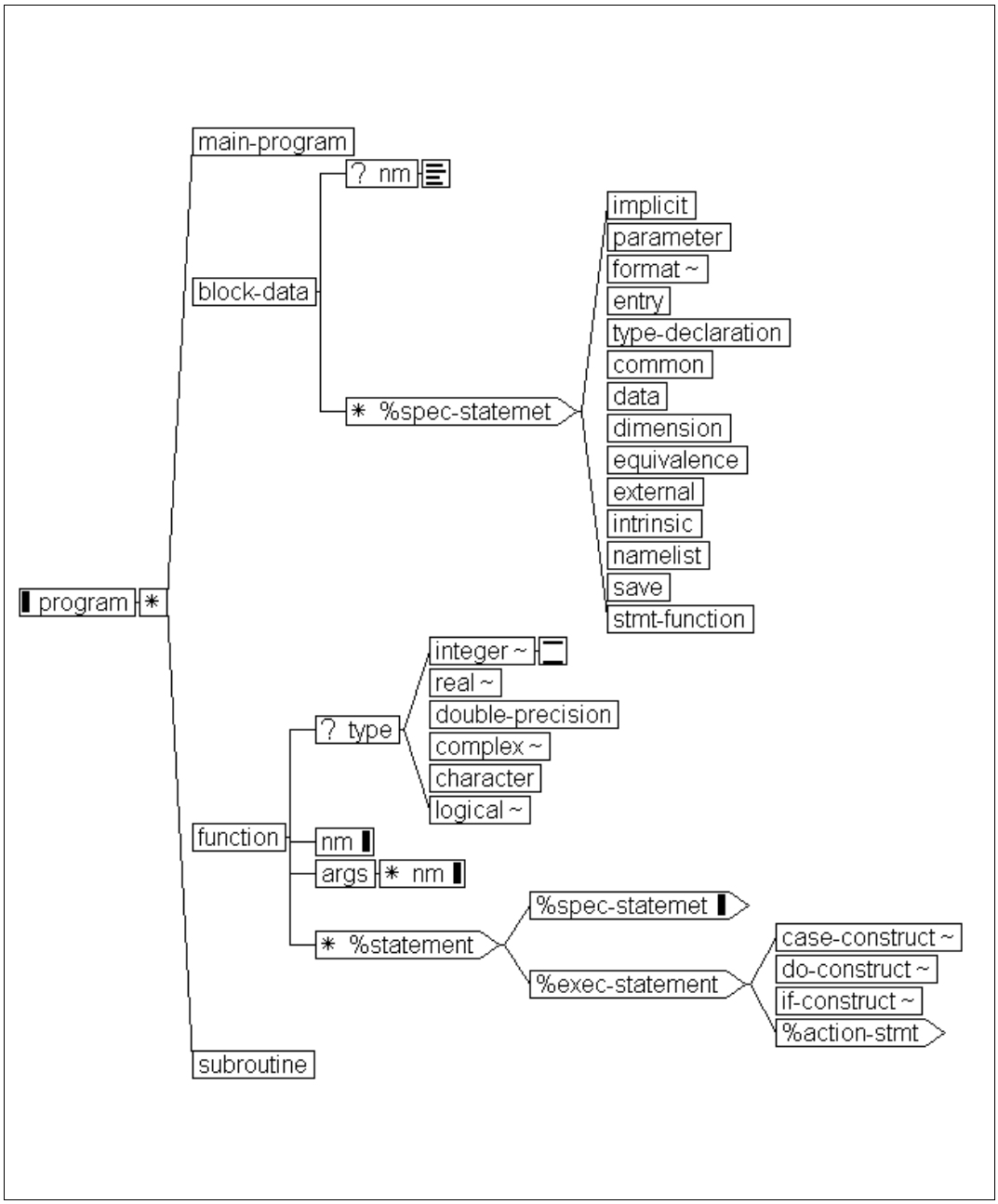


Figura 3 – Visão parcial da DTD para o fortran

4.4.2.1. *Tokens*

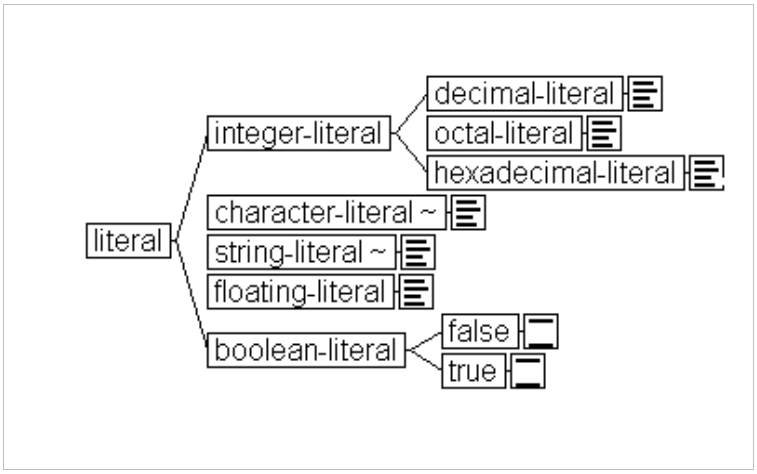
Os termos léxicos elementares, comumente denominados *tokens*, devem ser modelados com cuidado, pois são usados em qualquer programa que analise o código. Como são termos atômicos, há duas maneiras básicas de codificá-los: (a) elementos vazios com atributos e (b) elementos com texto. Por exemplo, o nome *count* pode ser codificado como (a) `<name a="count"/>` ou (b) `<name>count</name>`.

A primeira opção é claramente mais concisa, mas tem algumas desvantagens. A edição de atributos em editores XML apresenta algumas dificuldades, pois os esquemas de formatação como CSS atuam somente sobre o conteúdo. (Sim, é possível exibir um atributo como um texto gerado, mas não há como editá-lo interativamente.) Além disso, há uma certa deformação conceptual — o valor do atributo (como chamá-lo?) passa a conter o valor do nome.

A segunda opção, apesar de mais longa, é a naturalmente usada. Como os *tokens* aparecem muitas vezes no documento, é conveniente usar nomes curtos (por exemplo, *nm* para nomes e *lit* para constantes literais), o que alivia o ônus de usar dois marcadores ao invés de um único. Como veremos adiante, o que mais aumenta o tamanho do arquivo gerado é o número de níveis hierárquicos, e não os nomes dos elementos. Esta opção permite um uso mais racional dos atributos, por exemplo, para conter âncoras de documentação e informação gerada por processamento posterior do documento. Por exemplo, `<name type="int">count</name>`.

Outra aspecto envolve o tipo dos *tokens*; se há várias definições similares, devo usar um elemento único e opcionalmente distingui-los por um atributo, ou devo criar tantos elementos quantos forem os tipos? Bem, se o número de tipos é muito grande ou indefinido, não há opção senão usar um único elemento. Em alguns casos, entretanto, quando a distinção entre alguns poucos tipos é importante, pode

ser bom criar elementos distintos. Por exemplo, nas DTDs do TXL e do fortran, há um só elemento para representar literais (`lit`) e os tipos diferentes (*charlit*, *stringlit*, *number*, etc.) são distinguidos pelo atributo *t*. O conjunto de valores possíveis para *t* varia de acordo com o sistema de tipos da linguagem. No caso do ISO-C++, seria possível criar fidelidade máxima à gramática original, discriminando os tipos de constantes literais em vários elementos:



Quanto maior o número de elementos, mais simples as expressões para acessar elementos específicos, porém o custo disso é muito alto: o acréscimo de vários níveis representa um arquivo bem maior, além de dificuldade para edição manual. Por exemplo, em TXL uma constante numérica seria codificada como

```
<lit t='number'>5</lit>,
```

enquanto que no modelo C++ acima a mesma constante (um único caráter no arquivo original) seria convertida em

```
<literal>
  <integer-literal>
    <decimal-literal>5</decimal-literal>
  </integer-literal>
</literal>
```

Salta aos olhos o exagero da marcação, e a tentativa de usá-la para capturar o tipo do literal é infrutífera: para deduzir o tipo real (no exemplo, *int*) é preciso

analisar o conteúdo textual e verificar se não há um sufixo modificador (`U`, `u`, `l` ou `L`). Por essa razão, e também por dificuldades na análise léxica com TXL, a DTD final para o C++ acabou usando também um elemento *lit* genérico.

A definição dos elementos básicos pode ser reaproveitada de uma linguagem para outra se a definição for maleável o suficiente para deixar detalhes (como os sufixos acima) no conteúdo textual. Embora o grau de reúso dos gabaritos dependa muito mais do resultado desejado, o uso de nomes semelhantes torna o código XSLT mais claro. Há de se tomar cuidado, todavia, para não ignorar diferenças básicas entre as linguagens sendo processadas. Por exemplo, em fortran não existe distinção entre literais do tipo caráter e cadeia de caracteres (chamados usualmente de *charlit* e *stringlit*), são considerados apenas constantes de comprimento diferente. Se o nome do elemento (no caso, `lit`) pode ser aproveitado, os valores do atributo `t` (usado para guardar o tipo) devem ser diferentes, para evitar tratamento impróprio.

O efeito colateral da substituição dos delimitadores por marcadores já pode ser vislumbrado aqui: não é necessário armazenar as aspas ou apóstrofos que delimitam cadeias de caracteres.

4.4.2.2. Seqüências

A maioria das seqüências e listas previstas na gramática pode ser substituída por um elemento continente. O conteúdo nas gramáticas é comumente representado por um não-terminal genérico, que posteriormente será desmembrado em cada uma das produções permitidas. Em nossos modelos, esse não-terminal intermediário pode na maioria das vezes ser eliminado, bastando indicar todos os subelementos possíveis. Por exemplo, os termos iniciais da gramática fortran que usamos na seção 6.2 são:

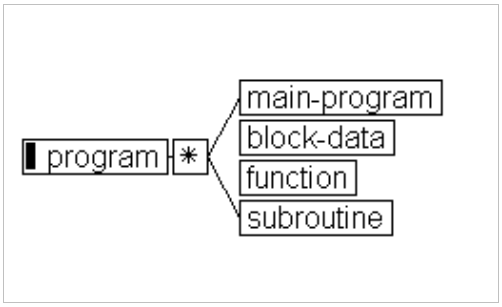
```
define executable_program
   [program_unit+] [line_comment*]
```

```
end define

define program_unit
   [main_program] | [external_subprogram] | [block_data]
end define

define external_subprogram
   [function_subprogram] | [subroutine_subprogram]
end define
```

Como os comentários não são considerados parte do fonte, a marcação equivalente pode ser simplificada para:



ou, na notação textual da DTD:

```
<!ELEMENT program
          (main-program | block-data | function | subroutine)* >
```

Os termos intermediários *program-unit* e *external-subprogram* foram omitidos, e a cardinalidade da seqüência foi alterada de 1…∞ para 0…∞, para acomodar arquivos vazios[8]. Estes não são programas fortran válidos, mas podem ser tolerados para facilitar a edição.

Listas com os elementos separados por vírgulas ou outros delimitadores são análoga e facilmente transpostas.

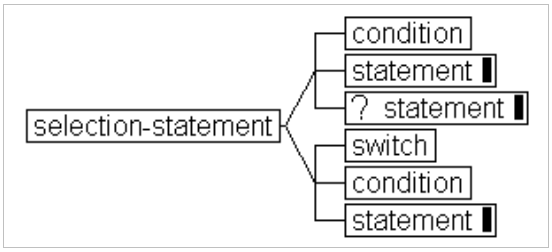
[8] Quando se fala de um arquivo XML vazio, subtende-se um elemento principal vazio, e não todo o arquivo, já que a omissão da declaração XML torná-lo-ia inválido.

4.4.2.3. Blocos

Os maioria dos termos que definem macroestruturas, como blocos, classes, comandos, etc. são facilmente transpostos. As palavras-chaves, que agem como delimitadores, são substituídas pelos marcadores e as produções por especificações de conteúdo correspondentes (que podem ser recursivas). Dois cuidados devem ser tomados: (a) é preciso fatorar os termos comuns de produções alternativas, pois a especificação XML exige comportamento determinístico dos modelos e (b) atentar para construções sintaticamente semelhantes, mas com semântica distinta. Por exemplo, o não-terminal *selection-statement* da gramática C++, que em TXL é definido como

```
define selection_statement
   'if '( [condition] ') [statement]
   | 'if '( [condition] ') [statement] 'else [statement]
   | 'switch '( [condition] ') [statement]
end define
```

foi transposta para a DTD como



Observe-se que:

- as duas produções *if* foram fatoradas, e a parte correspondente ao *else* representada por um elemento opcional;
- praticamente todos os delimitadores, incluindo palavras-chaves, foram descartados, à exceção da palavra `switch`, preservada sob a forma de um elemento vazio, que distingue o bloco *switch-case* dum simples *if* .

Note que do ponto de vista da gramática, as produções para um *if* simples (sem *else*) e um *switch* são idênticas. Mas é essencial distingui-las, pois um aplicativo precisa saber qual semântica usar ao interpretar os termos *condition* e *statement*. A tentação de unificar os ramos acima é grande. Não poderíamos distinguir os blocos pela presença de elementos *case* nos elementos *statement*? A resposta é: não. Embora um bloco *switch* sem os *cases* possa ser classificado como um objeto grotesco, é perfeitamente válido gramaticalmente. Tentar restringir a gramática original da linguagem, impondo restrições adicionais, é algo que não deve ser feito através das DTDs, mas através de ferramentas de validação. Num nível mais elevado, essas ferramentas permitem selecionar o conjunto de regras que desejamos aplicar, que pode variar de projeto em projeto. Como vimos anteriormente, é impossível construir uma DTD que reflita perfeitamente a gramática de uma linguagem complexa e dependente de contexto como C++. O que nosso modelo deve fazer é garantir que a estrutura hierárquica da sintaxe seja preservada, de preferência com o mínimo de marcação.

4.4.3. Migrando sem traumas

Um código fonte em formato texto não é totalmente desprovido de estrutura: palavras, linhas e colunas carregam informação que pode ser relevante para o compilador e para o leitor humano. Um compilador fortran interpreta os caracteres de forma diferente conforme a coluna onde se encontra; o pré-processador C reconhece linhas de continuação através de caracteres especiais; praticamente todas as mensagens de erro e advertência vêm acompanhadas da provável localização no texto. Para um ser humano, o arranjo dos termos numa página é de enorme influência na legibilidade do código.

Mas o documento XML possui uma estrutura diferente. A árvore do documento não reflete nenhuma disposição espacial em particular, mas a estrutura lógica do código. Como essa visão diferente do documento afeta a codificação?

4.4.3.1. Caracteres

As linguagens naturais são muito exigentes quanto aos caracteres utilizados. Os alfabetos português e inglês, por exemplo, são diferentes[9]. Surpreendentemente, as linguagens de programação também variam quanto ao conjunto de caracteres permitido. Algumas especificações ditam claramente quais os esquemas de codificação válidos, outras fazem referência aos sistemas operativos subjacentes, outras simplesmente ignoram o assunto. XML cai no primeiro grupo, e impõe um controle rígido de quais caracteres são permitidos e como indicar o esquema de codificação vigente. Não é de se espantar que tais regras entrem em conflito com as das linguagens de programação.

O primeiro aspecto refere-se ao conjunto de caracteres e a codificação utilizada. O conjunto de caracteres válidos em qualquer arquivo XML é o conjunto universal UCS definido pelo padrão Unicode. A codificação utilizada, entretanto, pode variar. O esquema UTF-8 é o presumido na ausência do atributo `encoding`, e o esquema alternativo UTF-16 é necessariamente suportado. Não é incomum, porém, que os processadores XML suportem diversos outros esquemas, em particular os da série ISO-8859. O difundido esquema ISO-8859-1 (Latin-1) é reconhecido por praticamente qualquer ferramenta XML. Como este esquema é suportado em vários sistemas operativos, e por quase todos os compiladores, é recomendado que seja utilizado ao codificar os fontes, para garantir interoperabilidade com outras ferra-

[9] O alfabeto inglês inclui os caracteres latinos ‘k’, ‘w’ e ‘y’, ausentes do português, e este usa diversos sinais diacríticos que os bretões ignoram.

mentas. No futuro, é provável que UTF-8 passe a ser a melhor opção, pois evita que tenha-se que lançar mão de seqüências de escape para representar caracteres não ASCII.

XML foi projetada com uma séria de restrições de ordem prática, que fazem com que os processadores sejam simples e rápidos. O determinismo do mecanismo de análise léxica á uma delas: ao ler um caractere, o processador sabe exatamente o que fazer, sem ter que lançar mão de métodos mais sofisticados como *backtracking*. Por exemplo, fora de seções *CDATA* (cujo conteúdo é sempre tratado *ipsis literis*), o caráter '<' sempre introduz um marcador ou instrução especial (de processamento, comentário, uma seção *CDATA*, etc.). Analogamente, o ampersande ('&') introduz uma referência a uma entidade. Por esta razão, os caracteres '<', '>' e '&', quando usados literalmente, devem ser escritos usando referências, repectivamente `<`, `>` e `&`. O mesmo ocorre para aspas dentro de uma cadeia delimitada por aspas ou apóstrofos numa cadeia delimitada por apóstrofos (respectivamente `&quot;` e `&apos;`). Esses caracteres são comuns em linguagens de programação, e qualquer aplicativo que converta de formato texto para XML ou vice-versa deve estar atento. Isso não quer dizer que o texto das cadeias deva ser alterado. Por exemplo, em C, as aspas no interior de uma constante literal deve ser precedida de uma contrabarra ('\'), que funciona como caráter de escape. De outro modo, a presença das aspas seria interpretada como o delimitador final da cadeia. Apesar da contra-barra não ser significativa em XML, não se deve sucumbir à tentação de suprimi-la. Isso causaria confusão, pois as demais seqüências de escape (\t, \n, etc.) não poderi-am ser convertidas da mesma forma. Mantê-las no formato original não só facilita o enleio e a edição, mas garante fidelidade à linguagem de programação. Isso é essencial no caso de documentos mistos — cada linguagem tem regras diferentes para caracteres especiais.

Um outro aspecto peculiar de SGML/XML é o tratamento dos caracteres em branco: tabulação, quebra-de-linha e espaço. Antes de tudo, é interessante notar que estes são os únicos caracteres de controle permitidos num arquivo XML. Qualquer outro caractere, como tabulador vertical, quebra-de-folha, etc. é ilícito e invalida o arquivo. Isso previne interpretações dependentes do equipamento ou da plataforma. O tratamento dos caracteres em branco segue regras bem rígidas, e pode ser controlado por meio de um atributo especial pré-definido, de nome `xml:space`. Seu valor presuntivo é `default` (*sic*), e espera-se do processador que suprima brancos redundantes, a saber: brancos consecutivos ou localizados no início e no final de um trecho marcado. O outro valor possível é `preserve`, e neste caso, todos os caracteres em branco são preservados. Este valor deve ser sempre usado ao transcrevermos trechos do código que contenham constantes literais, pois neste caso os espaços são sempre significativos.

E quanto ao restante do código? Será possível preservar os espaços e o estilo original ditado pelo usuário? Certamente isso vai de encontro à prática de controlar a aparência do código, e invalida qualquer ferramenta de formatação automática. Será que o usuário deseja realmente preservar seu estilo de formatar? No nosso entender, o que é importante não é preservá-lo, mas tornar o código *independente* do estilo. Destarte, o usuário não só tem a liberdade de escolher o estilo que mais lhe agrada, mas está livre da tarefa enfadonha e inglória de contar espaços e ajeitar linhas. Mesmo sem usar recursos avançados de formatação, com apenas uma folha de estilos CSS, o resultado é superior a qualquer arranjo possível em modo texto. Se usarmos recursos sofisticados, como numa tecedura completa, é possível obter um material com qualidade de impressão muito além de qualquer listagem saída de um *"pretty-printer"*.

Em nossos DTDs, assumimos o comportamento padrão (eliminação de espaços redundantes) em qualquer conteúdo textual, e nos casos onde a preservação dos

espaços é relevante (constantes literais, declarações de formato), explicitamos isso no elemento. Por exemplo, uma constante literal TXL do tipo *stringlit* e conteúdo "três espaços" é codificada como

```
<lit t='stringlit' xml:space='preserve'>três   espaços</lit>
```

Veremos adiante como esse mesmo problema da formatação inerente aos espaços pode afetar a criação e importação de comentários.

4.4.3.2. Linhas

A divisão do texto em linhas é essencial na análise léxica de muitas linguagens, principalmente as mais antigas, como cobol e fortran. Mesmo nas linguagens mais recentes, ainda há pontos onde as quebras de linha são significativas. Em java, uma cadeia de caracteres não pode conter uma quebra de linha: o programador deve optar entre usar uma seqüência de escape (neste caso assume-se que a cadeia contém um caráter quebra-de-linha) ou terminar a cadeia com aspas e continuar na linha seguinte (a quebra de linha fará parte do fonte, mas não da cadeia). A sintaxe da linguagem C identifica os comandos do pré-processador com linhas iniciadas pelo caráter '#' na primeira coluna; se o comando se estende por várias linhas, é mandatório usar uma contrabarra como caráter de continuação.

Ao criar um modelo XML para o código-fonte, devemos criar mecanismos para emular ou contornar estas dependências da estrutura de linhas e colunas. Uma cadeia de caracteres, por exemplo, é uma unidade sintática que deve ser armazenada como tal, e não secionada por razões estéticas. O mesmo acontece com linhas de continuação.

Quando pensamos nos compiladores e outras ferramentas, nossa postura tem que ser mais pragmática. Durante um período indefinido de tempo, tais ferramentas continuarão a emitir mensagens com referências a linhas e colunas no texto

original, e é necessário criar um meio de associar as linhas do fonte em ASCII aos elementos XML do hiperfonte. Alguns sistemas de programação letrada utilizam recursos do compilador, como a instrução *#line* do pré-processador C, para criar um vínculo com o hiperfonte: as linhas do código original são artificialmente numeradas para refletir a seqüência de elementos do hiperfonte. Esta solução, entretanto, não se aplica a outras linguagens. Uma solução mais geral é gerar, durante o processo de enleio, comentários que associem inequivocamente o texto a um elemento no documento XML. Isso pode ser feito criando uma expressão XPath cujo resultado seja o elemento sendo enleado. Por exemplo, uma regra de nome *func* em TXL pode ser identificada sem ambigüidade pela expressão `"//rule[nm='func']"`. O uso de atributos do tipo ID também pode ser considerado, pois gera expressões XPath muito simples; o padrão XML garante que os valores destes atributos são únicos num documento. Por exemplo, um elemento com um atributo do tipo ID com valor *f_134* pode ser identificado pela expressão `"id('f_134')"`, independente do nome do elemento, do atributo ou de sua posição relativa no documento.

4.4.4. Efeitos da linguagem de programação

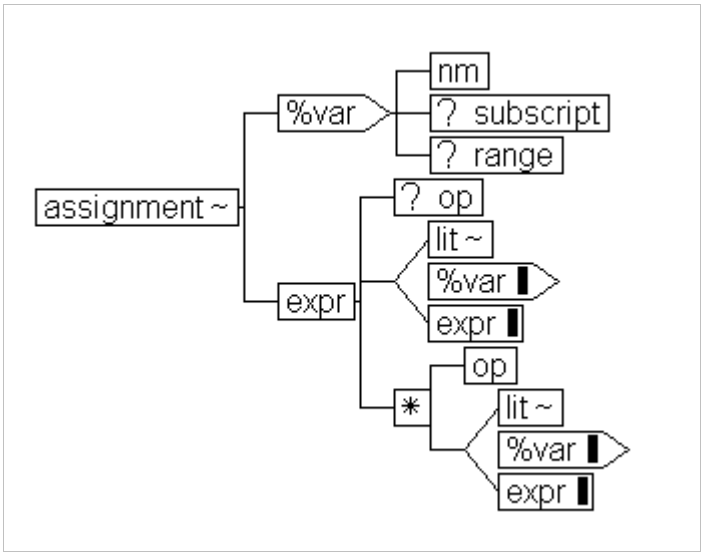
Toda linguagem de programação tem suas idiossincrasias — umas mais que outras. Uma linguagem com sintaxe simples, como lisp, seria facilmente transcrita para XML com um modelo muito simples. Gramáticas muito complicadas, como a do C++, apresentam uma série de obstáculos que podem inviabilizar o modelo. Vamos ressaltar os problemas mais comuns:

4.4.4.1. Expressões

É comum as gramáticas usarem uma série de termos intermediários para capturar a associatividade dos operadores em expressões. Se transcritos totalmente, esses termos oneram enormemente o esquema de marcação, já que a maioria das

expressões num código são simples (para serem legíveis). Veja na Listagem 2 na página nº 48 para um exemplo de uma expressão simples e dos termos gramaticais usados para derivá-la. Para evitar uma marcação tão pesada, aplicam-se reduções de ordem prática como as citadas na seção 4.4.2.3.

Nosso modelo para fortran, por exemplo, não distingue os operadores ou os termos de uma expressão. Em virtude disso, a marcação é simplificada. Por exemplo, o elemento `assignment`, que corresponde a uma atribuição, tem o seguinte modelo:



Observe-se que os parênteses são representados por um elemento *expr* no interior de outro. Também não há distinção entre uma chamada de função e o acesso a uma variável com índices, de acordo com o espírito da própria linguagem de programação.

4.4.4.2. Pré-processadores

A análise de algumas linguagens, em particular C e C++, se dá em duas etapas[10]. A primeira, conhecida como pré-processamento, é responsável pela interpretação e expansão de símbolos e macros, inclusão de outros arquivos por referência, e eliminação de comentários e outros blocos removíveis condicionalmente. A segunda envolve a compilação propriamente dita do código. Estas etapas representam, na verdade, duas linguagens com sintaxe distinta e características muito diferentes. Por exemplo, no pré-processamento linhas e caracteres em branco são significativos, enquanto na compilação eles são ignorados, à exceção dos contidos em constantes literais.

Ao escrever um modelo para este tipo de linguagem, há duas opções: (a) substituímos o pré-processador original por um equivalente XML, a saber um conjunto de entidades criados especialmente para o dialeto, e modelamos somente a linguagem principal (C/C++); ou (b) adaptamos o modelo para tentar capturar ambas as gramáticas, obviamente com restrições. Eliminar definitivamente o pré-processador não é possível, pois agride a prática dessas linguagens de programação: é raríssimo encontrar um programa C ou C++ que não use o pré-processador.

A primeira abordagem gera um modelo mais formalmente rigoroso e consistente, porém esbarra em uma série de restrições práticas:

- a especificação XML não inclui entidades parametrizáveis equivalentes às macros;
- não há como remover (`#undef`) uma entidade já declarada;

---

[10] O padrão ISO define na verdade várias etapas, a divisão em apenas duas é uma simplificação didática.

- o processador XML não tem como avaliar as expressões lógicas necessárias para os `#ifs`; só consegue fazer testes simples (`#iddef` e `#ifndef`).

Além disso, as especificações de entidades da XML não são facilmente editáveis com um editor comum. Na verdade, são raros os editores XML que têm suporte à edição de tais declarações.

A segunda opção é, portanto, a única viável no momento. O modelo que sugerimos no próximo capítulo impõe, em virtude desta amalgamação das gramáticas, algumas restrições no uso de símbolos e macros, mas nenhuma tão séria que comprometa o uso do modelo. Na verdade, são restrições de ordem prática que já se encontram em manuais de estilo de programação. O construtor do modelo deve ter em mente, porém, que o mecanismo de macros é muito flexível (segundo alguns até demais) e sua presença numa linguagem de programação pode perverter completamente a aparência final do programa, já que um símbolo pode ser substituído por qualquer coisa, mesmo que isoladamente inválido. Por exemplo, se escrevermos as linhas

```
#define A int main { return
#define B 0;}
```

`A`, `B` e `AB` serão programas inválidos, mas `A B` seria um programa sintaticamente perfeito, embora visualmente não tenha a mais vaga semelhança com forma de um programa em C. Emular esse comportamente através da gramática é impossível (se fosse viável, não haveria porque existir um pré-processador), e de utilidade questionável. Todavia no próximo capítulo veremos através de um exemplo que é possível conseguir resultados plenamente satisfatórios com um mínimo de prejuízo.

4.5. Modelando os comentários

Como vimos no item 4.2, um esquema que reúna comentário e código seria pouco prático. Optamos por usar esquemas distintos, adaptando-os para representar os diferentes níveis semânticos: o código segue um modelo hierárquico correspondente à sua sintaxe e a documentação, que é constituída de metadados, usa um modelo apropriado à descrição do primeiro.

O padrão RDF *(Resource Description Framework)*, citado na seção 3.1.4.1, é um mecanismo genérico para codificar metadados, e prevê mecanismos de extensão que nos permite criar vários vocabulários (conjuntos de propriedades) adequados a diferentes comunidades. A definição formal deste vocabulário é feita através de um esquema RDF, que é uma descrição — em RDF — das propriedades descritas e suas relações. Um esquema RDF usa um modelo orientado a objetos, que permite reutilizar e estender definições de outros vocabulários, através da especialização de termos. A definição de um esquema completo para descrever um código-fonte qualquer foge do escopo deste trabalho, e merece uma pesquisa dedicada. Limitaremo-nos a mostrar, usando um esquema bem simples, como a estrutura básica de um documento RDF destinado a descrever um código fonte em XML.

É oportuno ressaltar que o documento RDF pode ser embarcado, se necessário, no próprio documento descrito, provendo metadados para um processador de documentos. Essa prática é comum em documentos (X)HTML, sendo os metadados utilizados por mecanismos de busca direcionados por tópicos (em oposição aos que fazem pesquisa literal do conteúdo). No nosso caso, foram usados arquivos separados por pura conveniência: durante a importação de um código em formato texto, os arquivos com o código e a documentação são gerados em paralelo, sem a necessidade de fazer duas passagens ou mesclar o resultado. Pode ser interessante usar um

arquivo único, contudo, para facilitar a implementação de editores, que no nosso caso teriam que trabalhar sempre com os dois arquivos paralelamente. Isto pode ser vital se quisermos fazer transformações que manipulem simultaneamente código e comentários — a linguagem XSLT permite ler documentos múltiplos, mas o resultado da transformação é sempre um documento único. Para gerar mais de um documento, seria preciso uilizar transformações paralelas, ou lançar mão de extensões da linguagem[(11)], disponíveis em vários transformadores.

Nosso modelo prevê apenas um marcador, `description`, para os comentários. O vocabulário do Dublin Core contém vários elementos úteis para a descrição de *software*, e um modelo completo lucraria em usá-lo como base. O conteúdo do comentário é qualquer fragmento XML bem formado, podendo variar desde texto sem marcação (resultado, por exemplo, da importação de código em formato texto) até um fragmento XHTML com trechos em MathML e SVG embarcados. Um sistema completo de programação letrada pode ser criado, usando XLinks para vincular as descrições entre si e com o código.

Nosso modelo prevê também um atributo `type`, que tem origem na importação de arquivos (discutida na seção 5.1) e é usado para distinguir diferentes tipos de comentários.

Um arquivo típico pode ser visto na Listagem 3.

---

(11) As discussões, já em andamento, sobre as versões 1.1 e 2.0 da linguagem prevêem uma forma padronizada de gerar múltiplos documentos com uma única transformação.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<rdf:RDF
  xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
  xmlns:dc='http://purl.org/dc/elements/1.1'
  xmlns:lux='http://www.inf.puc-rio.br/lux/rdf/'>

<rdf:Description about='somefile.txl.xml'>
  <dc:format>text/xml</dc:format>
  <dc:language> x-txl </dc:language>
  <dc:source> somefile.txl </dc:source>
  <dc:type>Software</dc:type>
</rdf:Description>

<rdf:Description about='somefile.txl.xml#_2'>
  <lux:description>Seção de declarações</lux:description>
</rdf:Description>
<rdf:Description about='somefile.txl.xml#_2'>
  <lux:description xml:space='preserve'>Outro comentário sobre o
mesmo elemento, que possui id='_2'</lux:description>
</rdf:Description>

<rdf:Description about='somefile.txl.xml#//define[nm="expr"]'>
  <lux:description>Este tipo representa uma
expressão</lux:description>
</rdf:Description>

</rdf:RDF>
```

Listagem 3 – Um arquivo RDF típico

Podemos observar as seguintes características importantes:

- O recurso de espaços nominais é usado para distinguir, através de prefixos, os elementos definidos no padrão RDF, os definidos pelo vocabulário DC (Dublin Core) e os definidos no nosso sistema (Lux).
- O arquivo RDF nada mais é que uma seqüência de elementos `rdf:Description`, cujo atributo `about` indica o quê está sendo descrito. Este atributo é uma URI, e pode indicar qualquer recurso disponível.
- Alguns elementos do DC são usados para descrever o arquivo, e o elemento `lux:description` é usado para os comentários normais do programa.

Note, que apesar de termos optado em criar pares de arquivos fonte-documentação, isso não é mandatório: um arquivo RDF pode descrever múltiplos recursos em múltiplos arquivos. Um arquiteto de sistema pode optar por manter toda a documentação de um programa num único arquivo, como os antigos sistemas de programação letrada, mas como já mencionado, isso pode causar problemas de desempenho em sistemas de porte. O ideal, contudo, é que todos os documentos XML sejam armazenados num banco de dados especializado em XML, e neste caso a opção por um ou mais arquivos é irrelevante.

5. OPERANDO COM O HIPERFONTE

Nesta seção vamos analisar os vários tipos de transformações envolvidas na utilização de um hiperfonte que siga os modelos descritos na seção 4. Estas operações estão ilustradas na Figura 4. Algumas das ferramentas aqui descritas foram implementadas como exemplos concretos da viabilidade da arquitetura, e estão descritas com detalhes na seção 6.

Para implementar as transformações, faremos uso de duas linguagens de transformação: TXL, que opera sobre arquivos em formato texto e é usada no processo de importação de código, e XSLT, projetada especialmente para tratar documentos XML. Descrever ambas as linguagens foge ao escopo deste trabalho. Para uma descrição completa da versão 8 de TXL, a melhor fonte é o próprio manual do sistema [Cordy]. XSLT, porém, é uma linguagem muito complexa e repleta de detalhes, pois apresenta características tanto de linguagem funcional como de linguagem imperativa. A especificação em [XSLT] não é muito didática, e sugerimos o livro de [Kay], onde se encontra uma ótima discussão dos aspectos teóricos e práticos da linguagem, além de muitos exemplos e alguns padrões de projeto comuns em programas XSLT. Não obstante, procuraremos ressaltar no decorrer do texto os aspectos mais relevantes ao nosso problema.

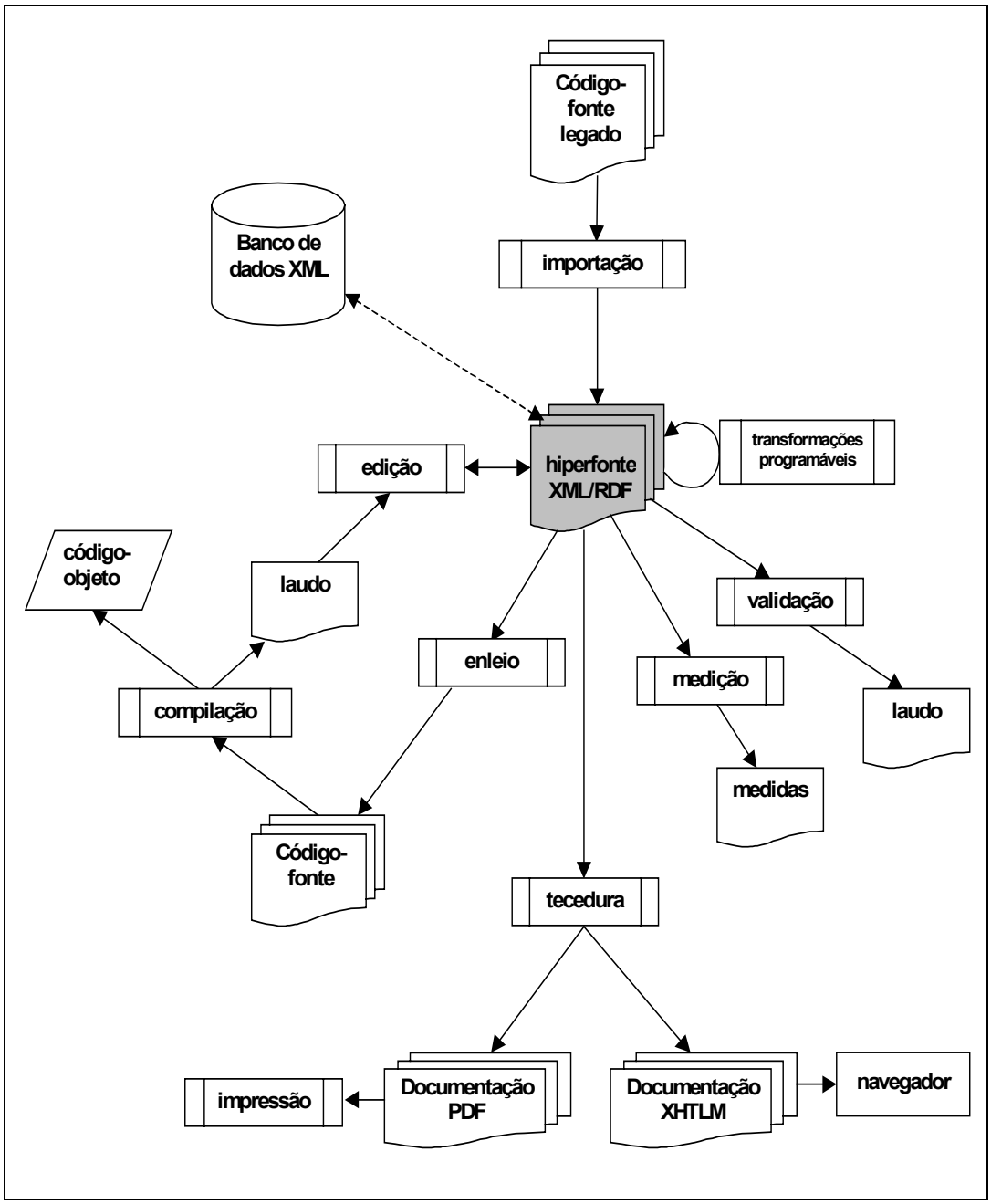


Figura 4 – Operações com o hiperfonte

5.1. Importando código legado

A tradução de código-fonte em formato texto para um modelo em XML é importante não só por uma questão prática, mas também estratégica. Um modelo

hermético, que não admite importação, tem muito poucas chances de ser bem aceito. Basta observar que todos os editores de texto comerciais possuem um vasto arsenal de importação, principalmente dos formatos de empresas rivais, mas não são tão pródigos em prover exportadores para os mesmos formatos. Uma das vantagens de usar XML é que ela foi especialmente projetada para interoperabilidade: o formato não só é facilmente transportável por rede, intrinsecamente neutro e independente de plataforma, mas pode ser convertido facilmente em outros através de transformações escritas em XSLT.

A etapa de importação é essencial para o aproveitamento do código já existente, mesmo se o hiperfonte não for usado para edição. Operar sobre o hiperfonte é bem mais barato que fazê-lo sobre o código original, não só porque o formato é uniforme mas porque muitas das ferramentas são distribuídas como *freeware*. Esta etapa foi a que consumiu maior tempo de pesquisa, no intuito de prover uma base sólida que permita o desenvolvimento das demais ferramentas usando como base de teste o vasto arsenal de programas legados.

5.1.1. TXL

A etapa de importação passa naturalmente pela análise sintática dos fontes originais. A escolha entre um analisador especializado na linguagem de programação escolhida ou um analisador configurável depende da disponibilidade das ferramentas e da facilidade em adaptá-las para gerar a saída desejada.

TXL é uma linguagem funcional híbrida, usada para transformações e desenvolvida por [Cordy] et alii na Queen's University, no Canadá. Foi escolhida por várias razões:

- seu interpretador está disponível no TecMF, cuja equipe forneceu inestimável apoio no uso da ferramenta;

- seu analisador léxico é muito eficiente e facilmente configurável, o que facilita seu uso mesmo em ambientes de produção;
- pudemos aproveitar algumas gramáticas já existentes;
- seu uso em processos de tradução já está bem amadurecido.

Em particular, o método da gramática achatada, descrito por [Felix] e posteriormente utilizada para traduzir fortran em HTML por [Amaral-00] presta-se a esse tipo de tradução, que pode ser feita em apenas um passe. Após a etapa de análise, a árvore sintática fica disponível na memória e podemos gerar o hiperfonte (e o arquivo RDF correspondente) à medida que caminhamos na árvore recursivamente da esquerda para direita. Esta seqüência pode ser subvertida quando a ordem de saída não é exatamente a original, mas no nosso caso isso raramente ocorre, pois o modelo XML foi criado de forma a refletir um fonte original.

5.1.2. Pré-processamento

TXL é muito útil para gerar o código final, mas nos deparamos com vários problemas no tratamento do código que exigiram um pré-processamento. Alguns deles têm a ver com os comentários e outros com limitações do processador TXL.

No decorrer do texto faremos menção a vários tipos de comentários. Denominamos comentário blocado ao delimitado por caracteres especiais e que pode se estender por várias linhas (e.g. `/*...*/` em C ou `{...}` em pascal) e comentário lateral ao que se inicia com uma seqüência especial e se estende até o fim da linha ou do arquivo, o que vier primeiro (e.g. `!...` em fortran ou `//...` em java). Também faremos distinção entre comentário associado (o que se descreve diretamente um determinado elemento do texto) e comentário narrativo (um trecho que narra uma seqüência de operações passadas ou vindouras intercalado no meio do código). Essa nomenclatura não é de uso geral, mas nos ajudará a explicar os tratamentos distintos.

5.1.2.1. Comentários

Há dois problemas que estão relacionados ao processamento simultâneo do código e dos comentários. O primeiro se refere à própria presença destes — ao invés de descartá-los já no início do processo, como os compiladores, nossos comentários serão processados junto com os demais elementos sintáticos. Neste caso, TXL exige que a posição dos comentários seja prevista na gramática, o que nos leva a um impasse. A maioria das linguagens de programação impõe pouca ou nenhuma restrição quanto ao posicionamento dos comentários. Fortran, por exemplo, admite comentários até mesmo no interior de identificadores e palavras-chaves. Prever na gramática todas essas possibilidades (algumas bizarras) é impraticável, e nossa única opção é limitar a colocação de comentários às posições usuais observadas no dia-a-dia. Na declaração de uma função, por exemplo, é razoável esperar comentários antes ou depois da declaração, e até mesmo ao lado dos argumentos, mas não no interior das declarações de tipo ou dos nomes dos objetos envolvidos. Se lembrarmos que os fontes que desejamos importar são programas reais, construídos (na maioria das vezes) por pessoas normais, veremos que restrições dessa natureza passarão quase sempre desapercebidas. Naturalmente, trechos de código patológicos, que constituem casos de teste para um bom compilador, não serão aceitos pela gramática. Em todo o caso, a gramática e o programa de tradução podem ser facilmente modificados para incluir esquemas de comentário pouco usuais.

O segundo problema tem a ver com a associação dos comentários. Como os comentários originais não possuem, a princípio, marcações que indiquem a qual elemento se referem, uma importação simplória anotaria somente sua posição no texto, provavelmente indicando o elemento sintático precedente. A prática usual de programação, contudo, utiliza a posição dos comentários para associá-los visualmente aos elementos. Neste processo, a distância da fronteira dos caracteres ao texto

e sua endentação relativa são os parâmetros principais, e podem ser capturados por regras heurísticas que, embora não resolvam todos os casos possíveis, permitam deduzir a associação da maioria dos comentários.

A regra que usamos para implementar o nosso pré-processador é simples, e baseia-se numa associação por proximidade. Se um comentário está numa linha adjacente a um elemento sintático qualquer, sem nenhuma linha em branco no meio, consideramos que se refere a este elemento. Comentários laterais de linha adjacentes que iniciem na mesma coluna são tratados como se fossem um comentário blocado (esse estilo é muito comum em C++ ou java). Cada bloco é associado ao elemento localizado à esquerda, à direita, acima ou abaixo, em ordem de preferência. Se um bloco de comentários se encontra entre dois blocos de código, mas é separado deles por uma ou mais linhas em branco, considera-se que não se refere a nenhum deles: é um comentário narrativo. Sua posição é armazenada em relação ao código, mas não é associado a nenhum elemento em particular.

Convém lembrar que esta regra só consegue ser efetiva em virtude da restrição sobre as posições válidas para um comentário. Seria inútil identificar ou associar blocos em posições inválidas. Regras diferentes podem ser implementadas onde o estilo de programação é muito diferente, ou quando o código é tão mal documentado que não vale a pena qualquer tentativa de associação automática.

Para que o interpretador TXL possa distinguir os tipos de comentários e suas associações, um marcador é inserido no início do comentário. Um caráter indica qual a associação deduzida (a=anterior, p=posterior ou ?=desconhecida) e um inteiro indica a posição do comentário num bloco de comentários laterais agrupados logicamente. Com base nesse marcador, o programa de tradução criará um vínculo entre o comentário, já sob a forma de um elemento RDF, e o elemento correspondente no código XML.

5.1.2.2. Caracteres brancos e delimitadores

Algumas linguagens são notoriamente de difícil análise. Fortran, por exemplo, foi desenvolvida praticamente sem nenhum embasamento teórico, e apresenta uma série de problemas léxicos que foram eliminados em linguagens mais recentes, com ajuda da lingüística computacional. Uma das principais diferenças se dá no papel de caracteres brancos (espaços, tabuladores e quebras-de-linha) na gramática. Tradicionalmente, os caracteres brancos não têm outra função além de delimitar *tokens*, mas em algumas linguagens seu uso foge ao convencional:

- Em fortran, os espaços e tabuladores são completamente ignorados se não estiverem contidos numa constante literal, enquanto as quebras-de-linha são significativas e delimitam os comandos.
- No pré-processador C, os brancos são significativos em alguns casos[12] e as quebras-de-linha delimitam as definições.
- Em perl, a tabulação (endentação) é significativa.

TXL foi desenvolvida para ser usada com linguagens tradicionais, e não possui mecanismos para analisar linguagens onde caracteres brancos são significativos. Esta deficiência exige a inclusão durante o pré-processamento de marcações especiais nos pontos onde seja necessário reconhecer esses caracteres. Por exemplo, ao pré-processar arquivos C ou C++, todas as linhas com comandos do pré-processador são finalizadas por uma seqüência especial para delimitá-los. Programas fortran exigem um processamento muito mais extenso, por causa da maneira singular como fortran ignora os espaços.

---

[12] Por exemplo, `#define REF(x) *x` define uma macro de nome *REF* com parâmetro *x*, mas `#define REF (x) *x` (note o espaço antes do parêntese) define apenas um símbolo *REF* como *(x)*x*, o que trará resultados completamente diferentes.

Note que essa deficiência no tratamento dos brancos torna TXL completamente imprópria para o tratamento de textos, sejam eles marcados ou não.

5.1.2.3. Caixa-alta e caixa-baixa

Algumas linguagens distinguem maiúsculas de minúsculas (e.g. algol, C++, XML) enquanto outras são-lhes insensíveis (fortran, pascal, HTML). Como TXL não possui suporte para linguagens insensíveis à caixa, o pré-processador é encarregado de normalizar a entrada. Não é viável escrever as gramáticas para tais linguagens de forma que aceitem todas as combinações possíveis de maiúsculas e minúsculas. Esta transformação inicial é saudável, pois uniformiza o aspecto do código: o uso de formas alternativas pode induzir ao erro.

5.1.2.4. Linhas longas

TXL possui um limite de implementação para o tamanho das linhas e cadeias de caracteres (254 caracteres). Linhas e cadeias muito longas são quebradas durante o pré-processamento e depois recompostas na fase de tradução.

5.1.2.5. Caracteres especiais e marcadores alienígenas

Para facilitar e agilizar a transcrição do conteúdo dos comentários para XML/RDF, os caracteres especiais proibidos em XML (<, >, &) são convertidos durante o pré-processamento nas referências de entidades correspondentes (< > e &). Se os comentários possuírem conteúdo já marcado, essa conversão deve ser desabilitada, para não destruir a marcação. Por exemplo, a ferramenta **javadoc** admite o uso de alguns marcadores HTML no interior dos comentários, e os utiliza na geração da documentação; ao importar código java com comentários neste estilo, deve-se inibir a conversão para não destruir as marcações HTML. Neste caso, porém, outras medidas devem ser tomadas para tornar a marcação compatível — de uma

maneira geral, marcações HTML não são compatíveis com XML. Uma opção simples e de fácil implementação constitui identificar os marcadores e convertê-los para XHTML.

Outras adaptações podem ser feitas no interior dos comentários para comportar marcadores usados por outras ferramentas. Por exemplo, os marcadores sugeridos por [Staa-00] e usados por [Braga] na implementação de Documentu podem ser facilmente convertidos para elementos RDF baseados no Dublin Core, o que preserva toda a metainformação neles contida.

5.1.3. Traduções e o método da gramática achatada

Após o pré-processamento, o código intermediário pode ser processado usando uma gramática TXL adequada à linguagem de programação. O processo de tradução usado é o descrito por [Felix] na implementação de LET, uma linguagem que roda sobre o interpretador TXL e facilita a escrita de programas de tradução. LET não foi todavia utilizada nesta pesquisa, porque o compilador LET ainda não apresentava, na época, uma implementação completa. Seria necessário incluir trechos em TXL para tratar detalhes como a conversão de cadeias de caracteres e a geração simultânea do co-arquivo RDF. O uso direto de TXL (com a ajuda de alguns minúsculos programas em C++) nos permite abordar a linguagem de forma reflexiva: a primeira linguagem que transcrevemos para XML foi... TXL!

Uma descrição detalhada da operação do programa tradutor constituiria uma longa narrativa. Em linhas gerais, ele consiste em uma séria de regras que são aplicadas a cada tipo de nó da árvore gramatical. Cada uma dessas regras toma uma das seguintes ações:

- Se este nó corresponde um não-terminal intermediário que não possui correspondente no modelo XML, ele é decomposto e as regras para os novos nós são invocadas na ordem apropriada.
- Se o nó corresponde a um não-terminal representado na DTD, o marcador inicial do elemento correspondente é gerado, seguido da aplicação das regras para o produto da decomposição, como acima, e do marcador final. Se o elemento pode ser alvo de um comentário, o marcador inicial é gerado com um ID, que pode ser usado numa eventual referência de um comentário.
- Se o nó corresponde a um *token*, seu correspondente XML é gerado, possivelmente com alguma alteração no conteúdo. (Por exemplo, remoção dos delimitadores ou conversão de caracteres especiais).
- Se o nó é um comentário ou bloco de comentários, uma regra especial é invocada que transcreve o conteúdo no co-arquivo RDF, ao invés de no arquivo XML principal. De acordo com a marcação inicial do comentário, inserida no pré-processamento, uma referência para o elemento sintático associado é gerada, usando seu ID.

O processo de caminhamento é recursivo, e termina quando o controle volta à regra correspondente à raiz da árvore gramatical. O resultado é um arquivo XML correspondente ao código original e um co-arquivo contendo os comentários na forma de elementos RDF, na mesma ordem em que apareciam no código.

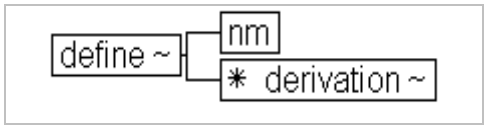
5.2. Enleio: regenerando o código

O enleio é o processo de gerar código compilável por um compilador tradicional, i.e., código em formato texto. Tal tarefa pode ser concretizada por uma trans-

formação XSLT cujo formato de saída é texto comum. A codificação de saída deve ser a esperada pelo compilador, na maioria dos casos ISO Latin-1. Isso correspondente ao seguinte cabeçalho:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:transform version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text" encoding="ISO-8859-1"/>
…
```

Uma transformação para enleio é tipicamente composta de uma série de pequenos gabaritos, cada qual responsável pelo enleio de um tipo de elemento. Como, na maioria dos casos, o código gerado para um elemento é independente do contexto, os gabaritos são ativados por casamento de padrões simples, e não por invocação direta. Por exemplo, o elemento *define* dum programa TXL é definido por



O gabarito correspondente de enleio é:

```
<xsl:template match="define">
   <xsl:text>define </xsl:text>
   <xsl:apply-templates select="nm"/>
   <xsl:call-template name="insert-label"/>
   <xsl:apply-templates select="derivation"/>
   <xsl:text>end define&#x0a;&#x0a;</xsl:text>
</xsl:template>
```

Os pontos importante a observar são:

- O gabarito é ativado pelo casamento com um elemento *define*.
- Elementos *xsl:text* são usados para gerar os trechos desejados. É possível inserir o texto diretamente no corpo do gabarito, mas o uso de *xsl:text* permite controlar mais de perto a inserção de espaços, o que é muito importante em arquivos-texto. O nosso gabarito de enleio para TXL visa produzir um código humanamente legível, e se preocupa em

endentar os elementos, inserir espaços onde necessário, etc. Note a dupla quebra de linha inserida com referências características após o texto `end define`.

- Elementos *xsl:apply-templates* são usados para ativar recursivamente os gabaritos dos elementos-filhos. Neste caso, ele é usado primeiro para o subelemento *nm* (que é único) e depois para os subelementos *derivation*, que são processados na ordem em que aparecem no hiperfonte.

Note, na linha central do exemplo, a invocação direta de um gabarito especial, chamado *insert-label*. Sua função é inserir no código um comentário lateral que contendo uma expressão XPath que referencia o elemento precedente. Isto pode ser usado para localizar o elemento responsável por um eventual erro de compilação. No nosso caso, por simplicidade, usamos o atributo ID do elemento (gerado na importação). Expressões independentes do atributo ID podem ser geradas na hora, se desejado, percorrendo a árvore gramatical na direção da raiz.

Nosso gabarito simples não enleia os comentários, pois consideramos que a função do código gerado é apenas alimentar o compilador. Se o intuito é regenerar completamente um código texto (e abandonar o hiperfonte), o gabarito de enleio deve carregar o co-arquivo RDF numa variável global usando a função *document* e, à medida que os elementos forem escritos, consultá-la para ver se há comentários associados. Encontrando-os, eles podem ser introduzidos no local apropriado. Um cuidado especial deve ser tomado se o comentário contiver texto marcado, para que o conteúdo não se deteriore.

5.3. Tecedura: formatando a documentação

Tecer a documentação a partir do hiperfonte significa não apenas apresentar a informação num formato agradável, mas compor índices e mapas que facilitem a navegação pelo código e compilar tabelas de métricas e outras informações importantes. Tal transformação requer gabaritos mais sofisticados, que coletem informação não só de várias partes do documentos, mas às vezes de vários arquivos simultaneamente. Há duas abordagens possíveis: a geração de uma documentação isolada e a formatação in situ.

A construção de uma documentação à parte, i.e., um conjunto de arquivos independente do hiperfonte original, é o caminho tomado pelos geradores tradicionais que operam diretamente sobre o fonte em formato texto. A diferença no nosso caso é que os elementos sintáticos relevantes já estão devidamente destacados no hiperfonte — basta organizar a informação de uma forma diferente. O formato de saída mais conveniente para nós é XHTML, porque pode ser reprocessado se necessário através de outras transformações XSLT. Por uma questão de retrocompatibilidade, pode-se desejar gerar a saída em HTML ou, se a intenção é obter manuais para distribuição, num formato mais próprio para impressão, como PDF, postscript ou TEX. De uma forma ou de outra, o destino final da documentação é via de regra um serviço de rede, que a torna disponível para os usuários finais.

Uma outra opção, que está se tornando viável à medida que cresce o suporte à família de padrões XML, é manter o código em XML e deixar a formatação para o cliente — no caso, o navegador. Um conjunto de folhas de estilo XSL é mantido no servidor junto com o código e a formatação é feita no próprio navegador que solicitou o documento. Esta solução tem a grande vantagem de manter a documentação

sempre atualizada, pois na verdade ela é criada e destruída a cada solicitação, e distribui a carga do processamento. Esta solução tem os seus poréns:

- São poucos os navegadores hoje em dia que conseguem realizar transformações XSLT sob demanda, e mesmo estes tem alguns problemas de conformidade.
- Como ainda não há navegadores com suporte a XSL-FO, é necessário usar um padrão de formatação diferente, como CSS. Para que o código em si possa ser visualizado adequadamente, pelo menos o nível 2 do padrão CSS é exigido, e também neste ponto os navegadores deixam a desejar.
- O processamento XSLT necessário é pesado, e fazê-lo nos navegadores clientes é antieconômico.

Estas três desvantagens podem ser superadas, todavia, se realizarmos as transformações num servidor dedicado, usando um só processador XSLT que produz e distribui páginas HTML/CSS sob demanda. Esta solução reúne o melhor das duas abordagens, inclusive permitindo diversas opções de formatação diferentes.

5.4. Editando o hiperfonte

A edição direta de arquivos XML implica no uso de editores especializados. Apesar de ser possível fazê-lo em modo texto, este modo de visualização é muito ineficiente. Apenas arquivos com marcação muito simples podem ser editados desta maneira.

A grande maioria dos editores XML existentes, que podem ser encontrados a mancheias na Internet, utiliza algum tipo de visão estruturada, arbórea ou tabular,

para auxiliar na edição. As interfaces gráficas usadas contém controles que permitem expandir ou contrair os nós da árvore do documento, de forma que somente os ramo de interesse estejam visíveis. Este modo de operação é particularmente útil na edição de tabelas e outros tipos de documentos com seções repetitivas, mas pouco ajuda no nosso caso. Um código-fonte possui estrutura complexa e heterogênea, e representá-lo como árvore ou tabela é completamente inapropriado.

O que um programador necessita é de uma visão ao qual esteja habituado, portanto similar ao velho texto, mas que permita tanto uma formatação mais sofisticada quanto o acesso à árvore do documento. Este tipo de funcionalidade só é encontrada atualmente nos editores mais sofisticados (leia-se: caros), que permitem associar folhas de estilo às DTDs. Através deste artifício, é possível visualizar os documentos numa visão semelhante à encontrada em editores de texto *WYSIWYG*. A diferença é que somente o conteúdo é armazenado no documento; a aparência de cada elemento pode ser controlada, mas é apenas um artifício para facilitar a edição. Como o formato não afeta o conteúdo gravado, é possível vários usuários editarem arquivos do mesmo tipo mas usando folhas de estilo diferentes. Por exemplo, um mesmo programa C++ pode ser visualizado usando diferentes folhas de estilo, contemplando estilos particulares de endentação.

No capítulo 6, veremos o resultado de algumas experiências feitas com um editor deste tipo, o XMetaL 2.1, produzido pela SoftQuad. Apesar do editor ter sido construído visando principalmente hipertexto XHTML e documentos do tipo narrativo, que seguem modelos como o DocBook, sua arquitetura é versátil o suficiente para adaptá-lo (com algum esforço, naturalmente) à edição de hiperfontes. A chave para uma boa adaptação está em aliar uma folha de estilos consistente com um conjunto de macros que tornem o trabalho de digitação o mais simples possível. A criação destas macros, contudo, está longe de ser uma tarefa simples, pois envolve uma dependência grande da DTD em foco.

Como discutido na seção 4.3.4, a utilização deste tipo de editor na edição de fontes só se dará se o formato em questão apresentar boas e concretas vantagens ao usuário programador. A simples adaptação do editor para operar sobre um "fonte virtual", uma visualização textual de um documento estruturado, não será uma dessas vantagens. Entretanto, o acesso que podemos ter, através de macros, à estrutura do documento, permitem realizar operações muito mais complexas, como:

- Busca e substituição sensíveis não só ao texto (usando o tradicional mecanismo de expressões regulares) mas também ao contexto.
- Criação e travessia automática de vínculos entre diversos pontos do documento, como o uso de uma variável e sua declaração.
- Busca automática de documentação pertinente a um nome qualquer, pela análise dos índices gerados na tecedura.
- Execução automática ou manual de qualquer transformação fornecida pelo usuário sobre o código, como por exemplo, uma seqüência de validação, enleio e compilação automáticas, seguidas, em caso de sucesso, da tecedura daquele arquivo.

Além de possibilitar um leque de opções muito mais rico para a automação de um editor, o uso de XSLT com linguagem padrão permitirá o intercâmbio de transformações complexas entre as comunidades de programadores, o que é dificultado hoje pela multidão de linguagens de macros existentes, cada uma específica de um editor.

5.5. Validação e controle

Um grande número de operações de validação pode ser realizada sobre o hiperfonte de maneira mais simples que sobre o código em formato texto. O fato da

árvore sintática estar representada quase que totalmente no documento, sendo o grau de semelhança dependente da profundidade da marcação, permite realizar facilmente testes sobre nomes, tipos, declarações, membros de estruturas, etc.

Transformações de validação têm via de regra uma estrutura muito simples:

- Um gabarito é criado contendo uma expressão de casamento que o ativa para as padrões de elementos que desejamos testar.
- No corpo do gabarito, o teste é realizado e em caso de falha uma mensagem é emitida.

Por exemplo, o seguinte gabarito testa se todas as classes no fonte C++ que derivam de outra possuem um destrutor virtual.

```
<!-- Matches all classes with base classes -->
<xsl:template
     matcht="//class-specifier[class-head/base-specifier]">
  <!-- retrieves class name -->
  <xsl:variable name="class-name"
                select="string(class-head/name)"/>
  <!-- check if a virtual destructor is declared -->
  <xsl:if test="not(string(member-declaration/
                function-definition
                [decl-specifier/function-specifier/virtual]/
                declarator/direct-declarator/declarator-id/
                id-expression/unqualified-id[tilde]/name)
                = $class-name)">
    <xsl:message>
      <xsl:text>Warning: class </xsl:text>
      <xsl:value-of select="$class-name”/>
      <xsl:text> has a base class but no virtual destructor!
      </xsl:text>
    </xsl:message>
  </xsl:if>
</xsl:template>
```

Este tipo de gabarito, que emite mensagens baseadas na verificação de testes e assertivas, é tão genérico que levou ao desenvolvimento de uma linguagem específica para validar estruturas por casamento de padrões, chamada Schematron (ver 3.1.3.3).

5.6. Medição: extraindo métricas

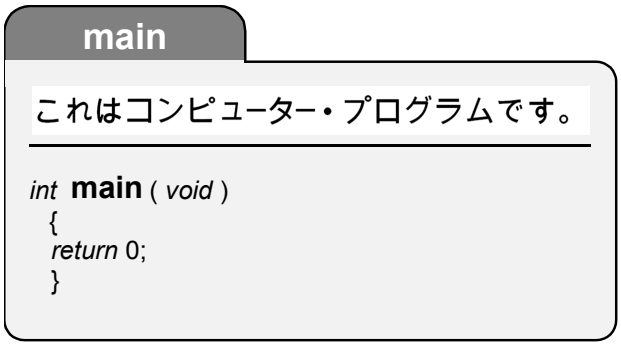
Embora algumas métricas tradicionais (e.g. LOC, *lines of code*) se apliquem especificamente a arquivos-texto, a maioria das métricas estruturais podem ser extraídas do hiperfonte com facilidade, usando padrões de casamento como os de validação. A única dificuldade aparece quando uma medida requer a abertura de vários arquivos simultaneamente (por exemplo, em medidas de acoplamento). Os atuais transformadores XSLT podem ter problemas se os arquivos são muito grandes, pois todos eles trazem o arquivo inteiramente para a memória antes de realizar qualquer operação, usando uma interface conhecida como DOM *(Document Object Model)*. Espera-se que futuras versões apresentem otimizações neste aspecto.

5.7. Transformações genéricas

Não há restrições no tipo de transformações XSLT que podem ser aplicadas sobre um hiperfonte. No caso do uso de XSLT se mostrar impróprio, não é complicado criar um programa usando um processador XML que trate os documentos manualmente. Se o custo de trazer todos os documentos para a memória inviabilizar o processamento, é possível trocar o analisador do tipo DOM por um outro tipo, conhecido como SAX *(Simple API for XML)*. Com SAX, o processador emite eventos para o aplicativo a cada passo da análise (como a leitura de um marcador inicial, um nó textual, um comentário, etc.), mas não traz o documento na memória. Este tipo de processamento é ideal para criar compiladores para o fonte.

Essa versatilidade contudo pode ser perigosa, pois não há garantias que o documento gerado pela transformação seja sempre válido. O uso da DTD para validar o resultado provê uma validação parcial, mas como vimos, ele não pode emular as gramáticas dependentes de contexto das linguagens de programação mais

populares. O uso integrado do Schematron (ver seção 3.1.3.3) pode ser uma solução para garantir transformações mais seguras.



6. EXEMPLOS

Para demonstrar a aplicabilidade da modelagem proposta, experimentamos com algumas linguagens de programação, criando transformações que operam sobre o hiperfonte de cada uma. O conjunto das DTDs, programas de importação e transformações em XSLT constitui um sistema aberto que foi batizado de Lux. O termo latino é uma alusão à clareza na prática da programação, mas pode também ser tomado como um acrônimo para ***l****ogiciário* ***u****sando* ***X****ML* [13].

Em Lux, os documentos são manipulados normalmente, em arquivos, e cada arquivo-fonte tradicional dá origem a dois arquivos XML, o primeiro contendo o código propriamente dito, seguindo uma DTD adequada à linguagem de programação, e outro contendo os comentários originais na forma de um arquivo RDF. Os arquivos gerados pelos programas de tradução conservam o nome original, acrescidos respectivamente das extensões *.xml* e *.rdf*. Quando mencionamos "código Lux"

[13] O galicismo *logiciário* não é tão popular quanto o termo *software*, porém é mais apropriado, pois programas são composições lógicas. E condiz com maquinário (*hardware*).

nos referimos ao par de arquivos. Note entretanto que esse par não forma necessariamente um tipo de entidade, pois a organização dos arquivos RDF não está vinculada à dos arquivos XML. Se for conveniente, por exemplo, podemos documentar o código inteiro usando apenas um arquivo RDF. Alguns exemplos de transformações, contudo, pressupõem a existência do par, apenas para efeito de demonstração.

Os programas de Lux foram escritos em três linguagens: TXL, na implementação dos tradutores; C++, para o pré-processamento e pequenos programas auxiliares; e XSLT para as transformações sobre XML. Além do interpretador TXL, usamos o compilador Microsoft Visual C++ 6.0 e o processador XSLT Xalan-C++ 1.1 da Apache Software Foundation. Contudo, procuramos nos ater ao ISO-C++ e à especificação XSLT 1.0, para que o código possa ser usado com outras ferramentas sem maiores problemas. As folhas de estilo para XML foram escritas usando recursos avançados de CSS2, e só serão corretamente visualizados em ferramentas com o suporte adequado, como o navegador Netscape Navigator 6 e o editor XMetaL 2.1, da SoftQuad.

6.1. TXL 8.0

Nosso principal exemplo envolve a própria linguagem usada para implementar os importadores de código. TXL pode não ter os recursos de sistemas mais avançados a criação de *parsers*, mas sua simplicidade e facilidade de uso tornam a linguagem atrativa para experimentações como Lux. Um sistema robusto para produção provavelmente utilizará analisadores especialmente sintonizados para as linguagens de programação suportadas.

6.1.1. Uma DTD para TXL

TXL também é um bom exemplo inicial pois possui uma sintaxe simples e livre de contexto, e portanto mais fielmente representada numa DTD. Nossa definição de tipo de documento para um fonte TXL está no arquivo *txl.dtd* (texto completo no apêndice) e pode ser visualizada na Figura 5.

Dado o caráter livre de contexto da gramática TXL, podemos observar que a DTD mimetiza-a de perto, com uma correspondência entre os elementos e os não terminais. A gramática utilizada como base (*txl.grm*, também no apêndice) foi adaptada da original que acompanha a ferramenta, para incluir os comentários e permitir a importação total do texto.

Os elementos *kw*, *nm* e *lit*, que representam palavras reservadas, nomes e literais (constantes de vários tipos), são usados para conter os *tokens* básicos e reaparecerão nas DTDs de outras linguagens, junto com o elemento *op*, que representa os operadores. Note que este último não aparece na DTD do TXL, já que a linguagem é funcional e não usa o conceito de operador, que não passa de uma facilidade notacional para melhorar a legibilidade. Em TXL, operações de qualquer natureza são sempre representadas por funções, mesmo que elas possuam nomes incomuns como '+' e '#'.

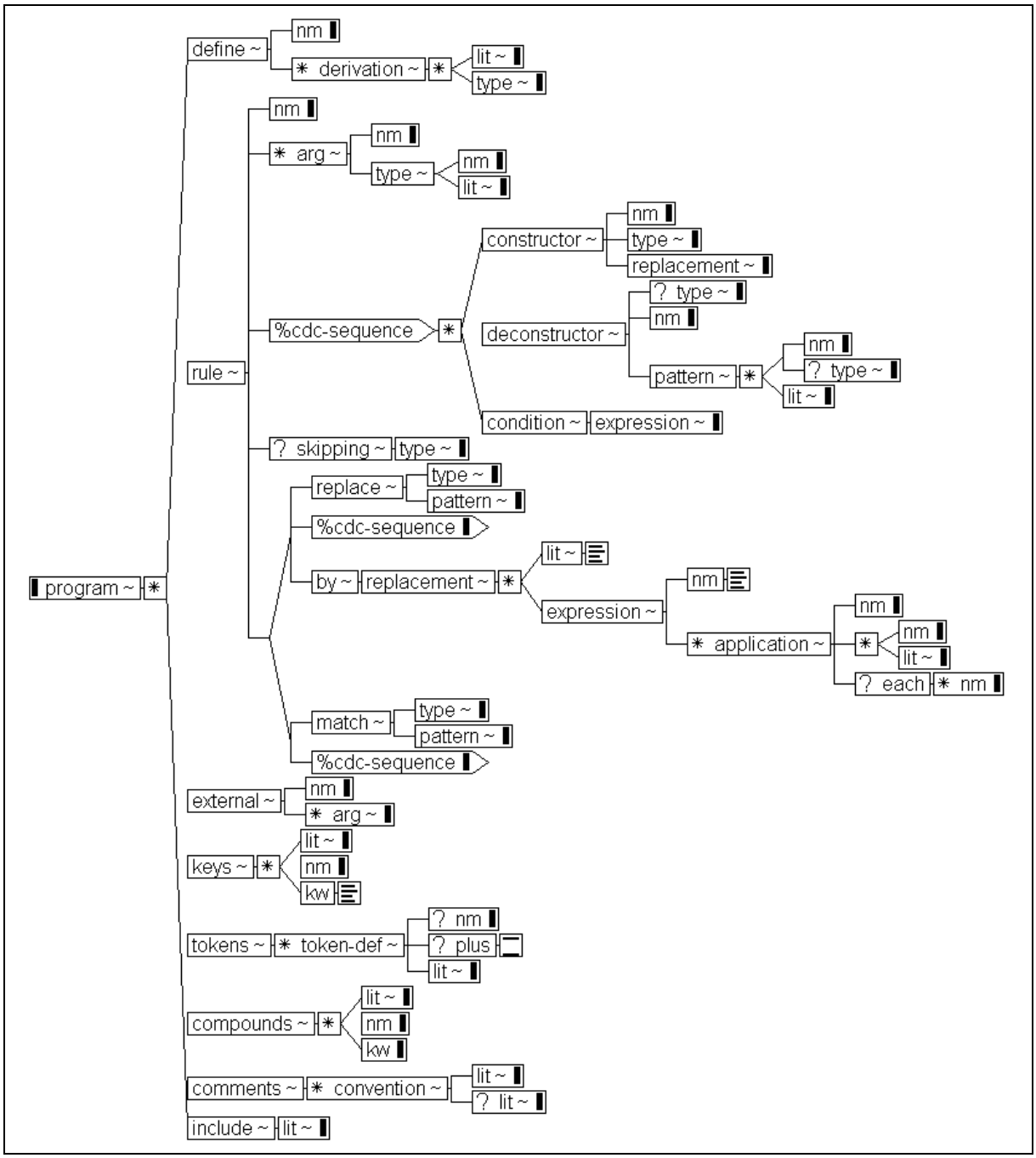


Figura 5 – A DTD para TXL

Outra característica que se repete nas demais DTDs é que as construções introduzidas por palavras-chaves são transpostas em ramos com um elemento-base equivalente. Estruturas morfologicamente semelhantes podem compartilhar o elemento-base, se conveniente. Neste exemplo, as definições de regras (`rule`), funções (`function`) e funções de busca (…`replace *`) são representadas pelo

mesmo elemento *rule*, e são distinguidas pelo atributo *apply*, que pode valer respectivamente "*once*", "*repeat*" e "*search*". Esta nova forma de representar das regras é até mais saudável, já que explicita a semântica um tanto obscurecida pela notação tradicional.

6.1.2. Importando TXL

Um programa tradutor foi construído para gerar um par de arquivos Lux (XML/RDF) para cada arquivo-fonte TXL. O programa *txl2lux.txl* usa o método da gramática achatada para gerar os dois arquivos durante um caminhamento recursivo na árvore gramatical gerada pelo *parser* TXL. Detalhes sobre o funcionamento deste tipo de programa podem ser encontrados em [Felix] e [Amaral-00]. O código possui descrições adicionais, e o fonte foi usado para testar a si próprio.

A tradução é feita em duas etapas. Primeiramente, o código original é pré-processado pelo programa cm.exe (o nome cm vem de *comment marker*), que analisa os comentários do fonte, e insere um marcador no formato "@#;" no início de cada comentário. # representa um número inteiro que indica se o comentário faz parte de um bloco: comentários em linhas contíguas iniciando na mesma coluna recebem números seqüenciais; em comentários isolados o número é sempre zero. O '@' representa um caráter que indica se o comentário está adjacente a um bloco de comandos da linguagem: 'a' (**a**nterior) indica que há código imediatamente antes do comentário; 'p' (**p**osterior) significa que o comentário é adjacente ao código subseqüente; e '?' indica que o comentário está desgarrado ou o programa foi incapaz de determinar uma direção conclusiva.

A etapa seguinte é a tradução propriamente dita, e as marcas introduzidas são usadas para criar *links* entre os blocos de comentário do arquivo RDF e os elementos do código aos quais se referem. Os tipos de elementos que podem ser

"apontados" por comentários são determinados pela DTD, que define um atributo do tipo ID em tais elementos. (Em XML, um atributo ID tem a garantia de unicidade dentro do mesmo documento.) Como só são gerados para os elementos semanticamente importantes, a associação automática funciona a contento, principalmente se o programador original foi consistente no posicionamento dos comentários.

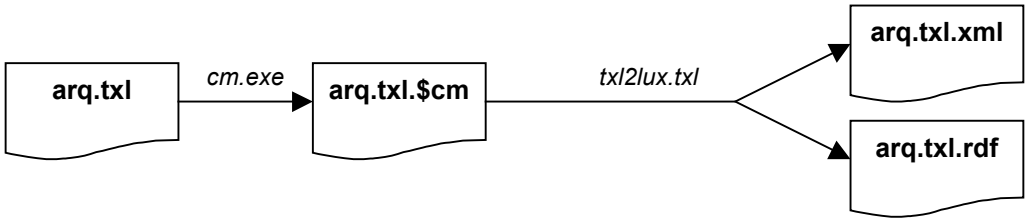


Figura 6 – Etapas do processo de importação

O código completo do sistema de pré-processamento é razoavelmente extenso, e não foi incluído no apêndice por economia de espaço. Sua arquitetura e funcionamento não contêm novidades, à exceção talvez do algoritmo usado para a associação heurística dos comentários. Caso o leitor se interesse neste aspecto da importação, o código completo pode ser obtido por solicitação ao autor. Pela mesma razão, o código gerado nos testes não foi incluído no apêndice: a listagem dos documentos XML são longas pouco legíveis sem uma folha de estilos apropriada.

6.1.3. Uma folha de estilos para código TXL

Usando CSS2, foi possível criar uma folha de estilos (*txl.css*, código no apêndice) que permita visualizar corretamente o hiperfonte TXL num navegador ou num editor. A formatação aplicada seguiu a forma usual de endentação usada no TecMF (o laboratório de métodos formais do Departamento de Informática da PUC-Rio), e o resultado pode ser visto na Figura 7.

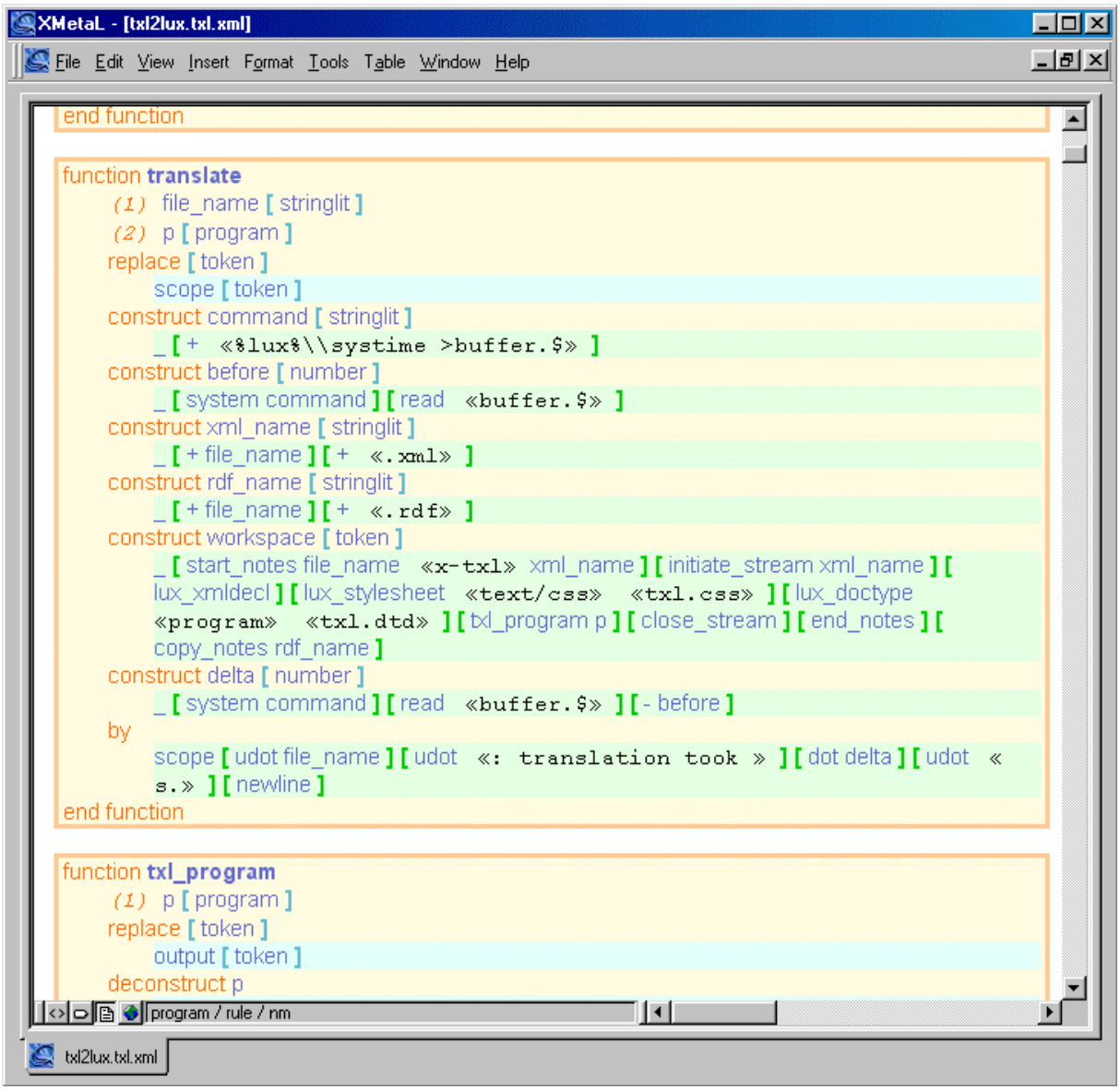


Figura 7 – Código TXL estilizado

A folha de estilos faz uso intensivo dos pseudoelementos *:before* e *:after* para introduzir os delimitadores usuais como colchetes e palavras-chaves, dando a impressão que o código contém os delimitadores originais e não somente o conteúdo das seções delimitadas. Por exemplo, aplicação de uma função de um argumento, que tem a forma `[func_name arg_name]` é codificada como

```
<application><nm>func_name</nm><nm>arg_name</nm></application>
```

A folha de estilos define o conteúdo dos pseudoelementos *application:before* e *application:after* como "[" e "]" respectivamente, o que gera na tela o padrão original. Note que CSS2 não permite alterar a posição relativa dos elementos, e portanto só é eficaz se a ordem dos elementos coincide com a ordem de visualização desejada e os "delimitadores virtuais" a serem gerados estiverem localizados na fronteira de um elemento XML. Caso um rearranjo dos termos seja necessário, uma folha de estilos CSS2 não será suficiente, e devemos recorrer a uma folha de estilos mais sofisticada, como XSLT gerando XHTML+CSS ou XSLT gerando XSL-FO. É bom lembrar porém que os editores especializados em XML costumam usar sempre CSS para formatar o fonte, e um cuidado adicional no projeto da DTD pode ser o suficiente para evitar o uso de recursos mais complicados.

A folha de estilos não precisa emular exatamente o formato original. Por exemplo, as múltiplas regras de derivação de um não-terminal num bloco *define* não foram separadas com a barra vertical original, mas usamos um contador para numerar as regras. O mesmo foi feito para os argumentos de uma função. Outras modificações para melhorar a legibilidade podem ser tentadas. Por exemplo, em pascal podemos substituir o sinal de atribuição '`:=`' por algo mais significativo como '←' a fim de evitar confusão com o operador de igualdade '`=`'. Em C, a troca de '`==`' por '`=`' nas expressões condicionais é uma fonte constante de erros.

O importante é que as folhas de estilos podem ser individualizadas. Cada usuário pode utilizar uma que seja mais do seu agrado, variando fonte, cor, esquemas de endentação, etc. O código é completamente independente da folha de estilos preferida, que, como a personalidade dos programadores, pode variar de espartana a psicodélica.

6.1.4. Uma folha de estilos para o arquivo RDF

A folha de estilos *rdf.css* permite visualizar diretamente o arquivo RDF criado num navegador. Para visualizar os arquivos do fonte e das descrições de forma sincronizada, seria necessário criar um terceiro arquivo através de uma transformação XSLT.

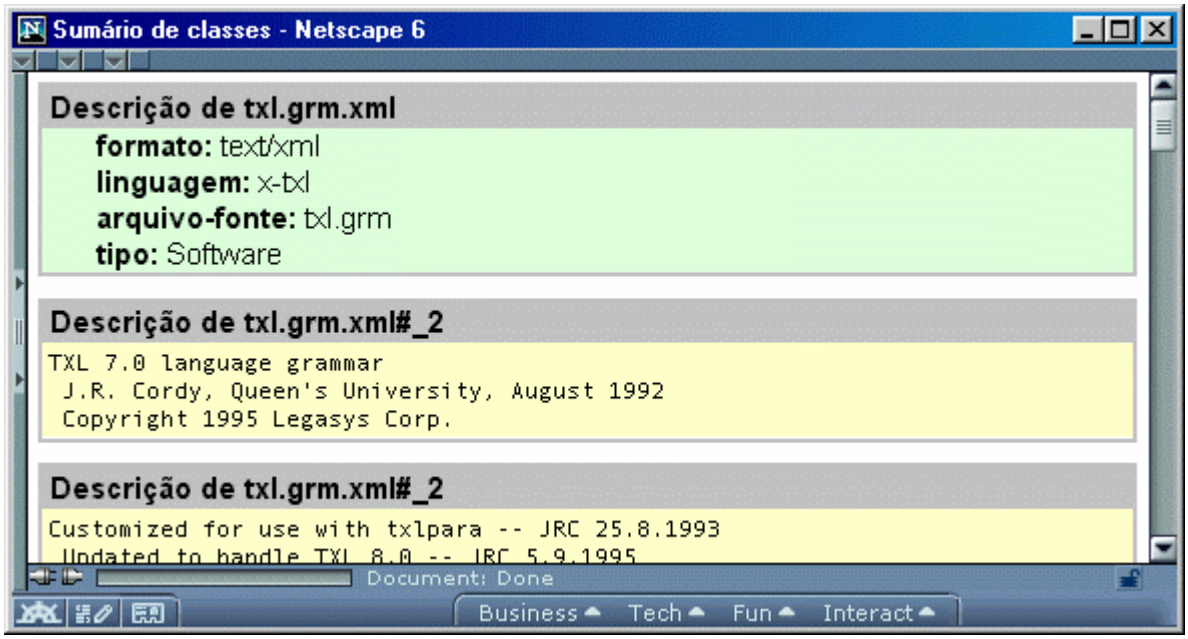


Figura 8 – Metainformação RDF estilizada

6.1.5. Enleio

Uma transformação XSLT simples é suficiente para um enleio preciso do código. O arquivo *txl_tangle.xslt* contém a nossa versão, que usa o arquivo XML para gerar um arquivo-texto processável pelo interpretador TXL. Note que os comentários não são incluídos, pois não são necessários se o código se destina apenas à compilação. Se desejarmos restaurá-los, contudo, o procedimento é simples. Na seção sobre C++, mostraremos como processar simultaneamente o código e os comentários.

A transformação de enleio foi usada para gerar código TXL compilável a partir dos geradores de programa descritos mais adiante.

6.1.6. Tecedura

A tecedura envolve criar uma documentação completa do código, compilando índices e tabelas adicionais aos comentários originais. Um programa completo de tecedura não apresenta dificuldades técnicas além da descritas mais adiante e tomaria um longo espaço, por isso nos abstemos de um exemplo completo de tecedura. Tal programa, contudo, é um dos produtos principais de um sistema como Lux. Uma documentação completa em XHTML pode ser colocada na Internet ou numa rede local, provendo um meio barato e prático para publicar a documentação dos sistemas. A maioria dos projetos de código aberto na Internet costumam disponibilizar algum tipo de documentação em HTML junto com as distribuições do produto.

6.1.7. Gerando programas

Uma das vantagens de manter código em XML é a facilidade de manipular o código automaticamente. Daremos dois exemplos simples do que pode ser feito manipulando o código TXL importado.

6.1.7.1. Capturando toda a árvore sintática

Durante a depuração de uma gramática TXL, é conveniente analisar a árvore gramatical gerada, que pode conter erros causados pela interpretação equivocada de construções ambíguas. Vasculhar uma árvore dentro do depurador é uma tarefa ingrata em virtude, já que essas árvores podem ser muito grandes. A transformação *txl_gentree.xslt* recebe como entrada uma gramática TXL qualquer, devidamente

importada para o formato Lux, e gera um programa TXL (também codificado em Lux) cuja função é fazer um *dumping* da árvore gramatical num arquivo XML bem formado. Este arquivo é uma representação arbórea equivalente à interna do interpretador TXL. Carregada num editor XML ou num navegador, podemos vasculhar esta árvore muito mais facilmente que do interior do depurador TXL.

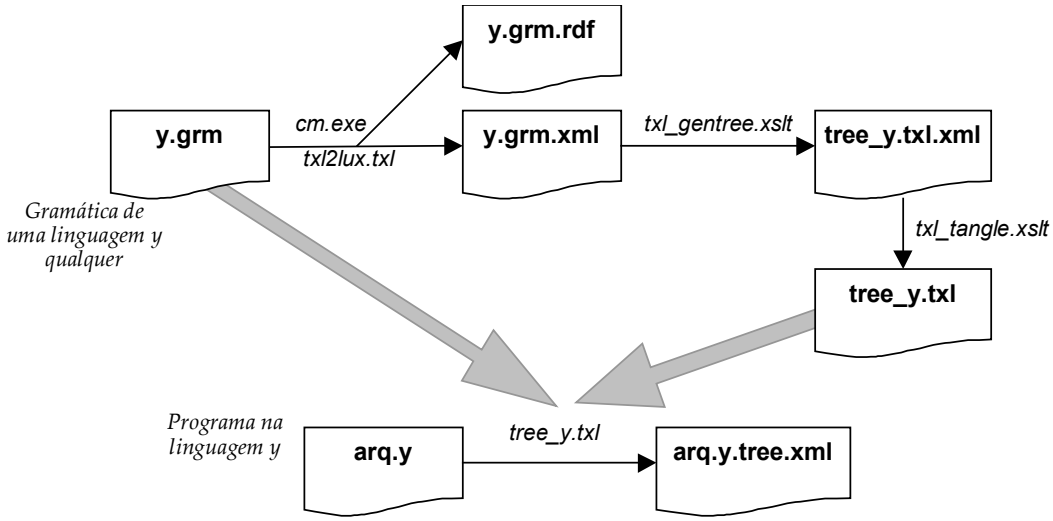


Figura 9 – Geração de um visualizador de árvore sintática

6.1.7.2. Gerando o protótipos de um tradutor

Usando a mesma estratégia, escrevemos uma transformação que cria não um gerador de árvores, mas o esqueleto de um programa tradutor. Partindo da gramática TXL de uma linguagem qualquer, são geradas as funções comuns a todos os tradutores e uma seqüência de funções para traduzir cada um dos não-terminais da gramática. O programa gerado não é completamente funcional, pois precisa ser ajustado à DTD escolhida para a linguagem, mas poupa o programador do enorme trabalho de gerar as funções de tradução, que seguem sempre o mesmo formato. Para se ter uma idéia do tempo economizado, o tradutor TXL→Lux tomou duas semanas de trabalho. Os tradutores fortran→Lux e C++→Lux, que possuem uma quantidade muito maior de funções, foram terminados em 2 dias cada, pois o trabalho se resumiu a adaptar as funções à DTD.

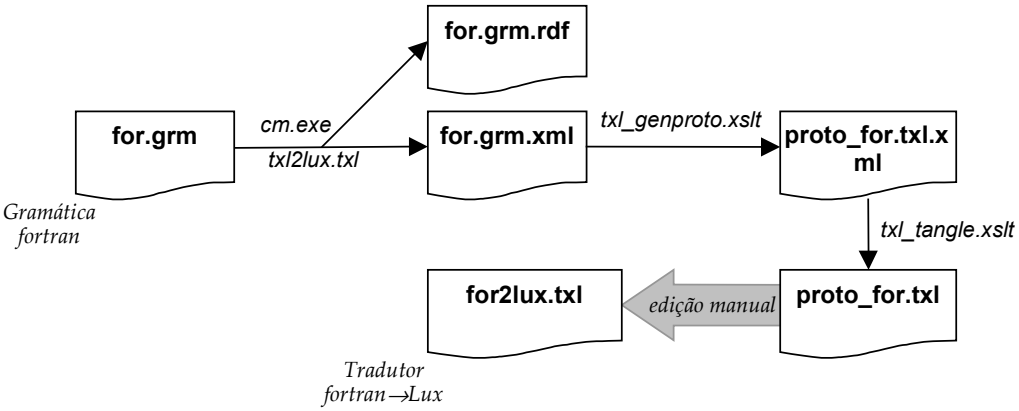


Figura 10 – Geração de um importador Lux

6.2. Fortran-77 estendido

Apesar do uso escasso de fortran-77 hoje em dia, em virtude da disponibilidade de compiladores fortran-90 e fortran-95, ainda se encontra muito código legado escrito em fortran-77, às vezes repleto de construções como `if`s aritméticos, que remontam à era dos computadores valvulados, antes do advento da programação estruturada. Por esta razão, achamos interessante incluir um tradutor para fortran. A manipulação automática desse código é altamente desejável, e a tradução para Lux viabiliza a implementação de tais transformações.

6.2.1. Uma DTD para fortran

A DTD que criamos para fortran é extensa, em virtude do grande número de comandos da linguagem, mas simples. A sintaxe do fortran possui poucas constru-

ções sensíveis ao contexto, como a dualidade vetor/função[14]. O sistema de tipos é simplíssimo e a sintaxe das expressões, embora extensa, pode ser simplificada, pois não é necessário reter a associação dos operadores. O texto completo da DTD está incluído no apêndice.

6.2.2. Importando fortran

A análise léxica de fortran-77 é problemática por duas características peculiares da linguagem: ela não tem palavras reservadas e ignora os caracteres brancos. Além disso, é insensível à caixa. Por exemplo, a linha

```
In te gerif , pro gram
```

equivale a

```
Integer if, program
```

e declara dois nomes (variáveis ou funções) do tipo inteiro.

Além disso, a posição dos caracteres nas linhas e colunas é significativa. Linguagens mais modernas possuem sintaxes mais regulares, com bases lingüísticas mais sólidas, evitando estruturas de difícil análise. TXL foi projetada com essas linguagens mais modernas em mente, e por isso entra em conflito com as regras do fortran.

Ao invés de usar um analisador léxico específico para fortran, adotamos uma solução menos ortodoxa. Um pré-processador específico foi criado para normalizar o fonte fortran, rescrevendo os *tokens* de forma a permitir o tratamento por TXL. Este

---

[14] Em verdade, um vetor pode ser encarado como uma função que mapeia o índice no domínio dos elementos nele contidos. O fato de fortran usar a mesma notação para vetores, funções e indexação de cadeias literais provavelmente não é intencional, mas condiz com o caráter matemático da linguagem.

método já havia sido utilizado com sucesso na construção de For-See [Amaral-00], e foi apenas ligeiramente adaptado para uso em Lux. O pré-processador fortran (*fpp.exe*) trata seqüencialmente cada comando do programa fortran, transformando-o da seguinte forma:

- expande as declarações `INCLUDE` (isso não é estritamente necessário, e pode ajudar ou atrapalhar as transformações — mais sobre o assunto nos exemplos sobre C++);
- pospõe o conteúdo das linhas de continuação à linha principal, garantindo que todo o comando esteja numa única cadeia;
- procura constantes holerith e as transforma no equivalente literal delimitado por apóstrofos;
- remove todos os caracteres em branco e normaliza o texto para caixa baixa (exceto no interior de contantes literais e `FORMAT`s);
- analisa a cadeia resultante para identificar as palavras-chaves iniciais que caracterizam o programa e insere espaços se necessário para separar *tokens* adjacentes;
- analisa contantes literais contidas na linha para converter as seqüências de escape fortran nas equivalentes TXL e as quebra em várias se exceder o limite de 254 caracteres do TXL;
- se a linha resultante for muito comprida, quebra-a em linhas mais curtas (a mesma limitação das cadeias literais vale para as linhas);
- coloca um ponto-e-vírgula ao final da linha para marcar o fim do comando.

Linhas de comentário são também normalizadas para usar sempre o mesmo caráter inicial ('!'). O resultado e gravado num arquivo de mesmo nome que o

original, mas com extensão *.fff* ao invés de *.for*. A partir daí, o processo é semelhante ao usado em TXL: o arquivo é processado pelo marcador de comentários (*cm.exe*) e depois traduzido pelo programa *for2lux.txl*. Note-se que a maior parte do corpo deste programa foi TXL gerado automaticamente pela transformação da gramática.

6.2.3. Enleio

Uma transformação de enleio para fortran seria muito semelhante ao de TXL ou qualquer outra linguagem, por isso nos abstemos de incluí-la neste trabalho. A única peculiaridade é que ao invés de produzir uma saída diretamente, o gabarito deve gerar o texto completo do comando numa variável temporária, convertê-lo numa cadeia de caracteres e só depois inseri-la na saída. Este tamponamento é necessário para evitar que o conteúdo da linha ultrapasse o limite da septuagésima segunda coluna: pode ser preciso usar linhas de continuação.

6.2.4. Processando múltiplos arquivos

O exemplo que usa o código fortran traduzido para Lux ilustra como processar diversos fontes simultaneamente. A transformação *for_projreport.xslt* lê uma arquivo XML simples que contém uma lista dos componentes de um projeto e analisa cada um em seqüência, criando um relatório. Por exemplo, o projeto do arquivo na Listagem 4 produz o relatório em XHTML mostrado na Figura 11.

```
<?xml version="1.0"?>
<project>
   <name>Tides</name>
   <file>two.for</file>
   <file>rhnry.for</file>
   <file>pvmol.for</file>
   <file>param.for</file>
   <file>matinv.for</file>
   <file>main.for</file>
   <file>liqfug.for</file>
   <file>flash.for</file>
   <file>ewequ.for</file>
   <file>dewpt.for</file>
   <file>value.for</file>
</project>
```

Listagem 4 – Um arquivo de projeto em XML

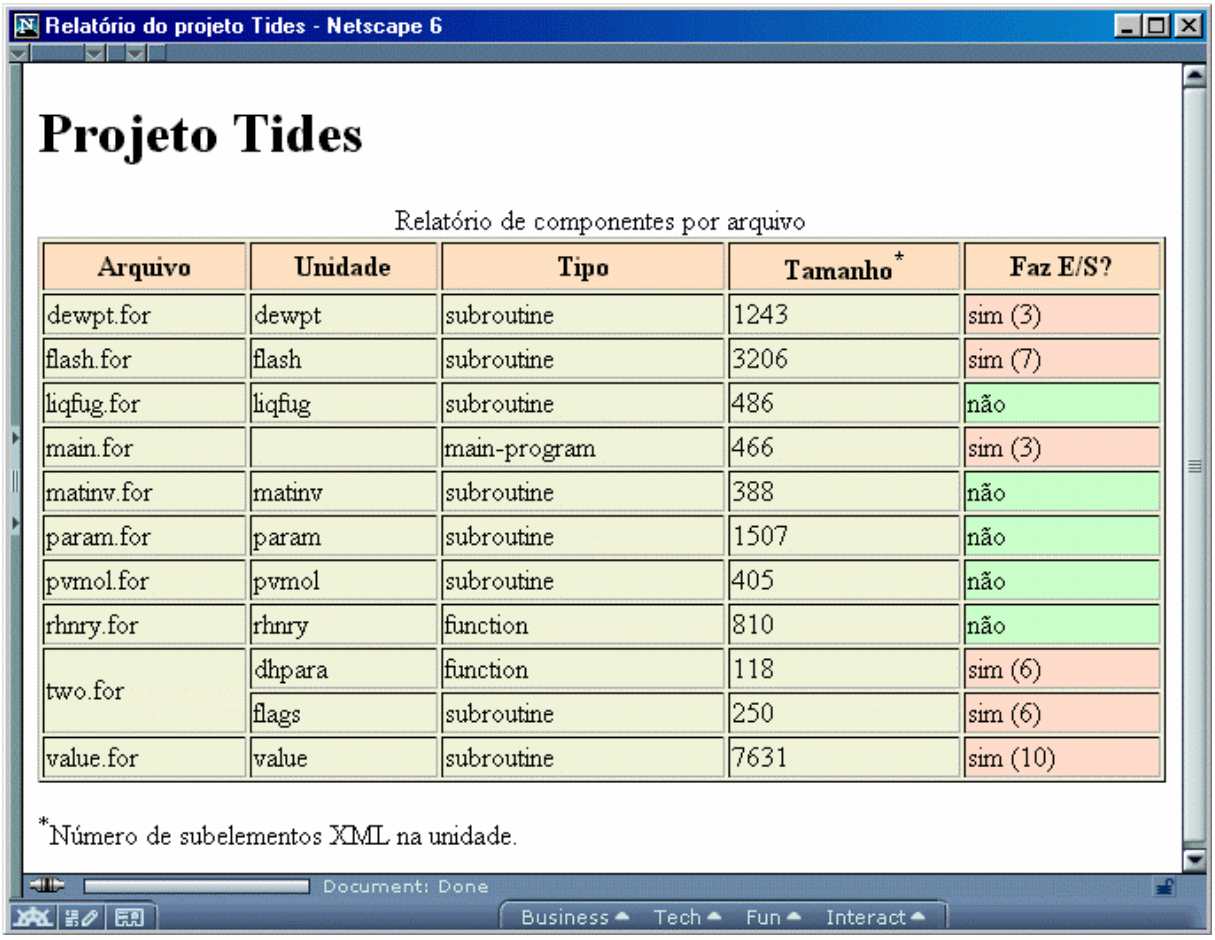


Figura 11 – Relatório para um projeto fortran

6.3. ISO-C++

C++ é uma linguagem que tem recebido inúmeras críticas por ser complexa demais. Sua enorme popularidade entretanto, é fruto de sua capacidade de suportar vários paradigmas de programação diferentes numa única gramática. Neste aspecto, pode-se compará-la a um canivete multifuncional: abriga diversas ferramentas, mas você terá dificuldades se tentar usá-las todas ao mesmo tempo.

A sintaxe do C++ está repleta de construções sensíveis ao contexto. Por exemplo, o trecho

```
t f(a);
```

no escopo global pode estar declarando:

- uma função *f* que recebe um argumento do tipo *a* e retorna um objeto do tipo *t*, ou
- um objeto *f* do tipo *t* iniciado com a expressão *a*.

O compilador só pode distinguir as duas opções conhecendo a prévia declaração do nome *a*, que é obrigatória. Se *a* for o nome de um tipo, *f* será uma função; se *a* for uma variável, *f* também o será.

Por esta razão, e pelo efeito transformador das macros (veja o item 4.4.4.2), a expansão das diretivas de pré-processamento (*#include*, *#if*, etc.) é vital para a análise plena do código. O que nos leva a um dilema ao portarmos o código para XML. Há vários problemas relacionados:

- Vários arquivos #incluídos são arquivos de cabeçalho padrões, como <*iostream*> ou <*algorithm*>. Esses arquivos, apesar de declararem interfaces padronizadas, estão repletos de estruturas particulares do com-

pilador utilizado, e que mudam de plataforma para plataforma. Portanto, mesmo que o código escrito pelo usuário seja 100% conforme ao padrão ISO, não será possível traduzir os arquivos do sistema. Uma solução seria criar uma versão ISO de cada arquivo de inclusão, e usá-los ao invés dos fornecidos com os compiladores.

- O volume dos arquivos de inclusão é quase sempre várias vezes o volume do arquivo principal. Expandi-los completamente atrapalharia grandemente a edição e introduziria uma grande redundância e desperdício de espaço, pois os mesmos arquivos de cabeçalho são incluídos repetidas vezes em cada um dos fontes.

Em suma, a presença de duas gramáticas diferentes (pré-processador e C++) inviabiliza uma transcrição completa da árvore gramatical. Se observarmos, porém, a prática de programação em C++ e a forma como os arquivos são escritos, veremos que a transcrição é possível, se ela se limitar à visão que o próprio programador tem do código. Vejamos:

- Quando o programador "#inclui" um arquivo, ele vê a diretiva de inclusão como mais um comando do arquivo. É como se ele estivesse escrevendo (e na verdade está) várias declarações de uma só vez.

- Quando se definem macros e símbolos, elas são declaradas de uma maneira que possam ser usadas no código de maneira a emular os elementos sintáticos tradicionais. Por exemplo, a definição de macros que incluem declarações usualmente não incluem o ponto-e-vírgula final, de maneira que seu uso no texto se assemelhe a uma declaração comum.

- A subdivisão do código em *tokens* e elementos sintáticos básicos pode ser feita mesmo sem as declarações serem resolvidas plenamente. No

exemplo acima, *a* pode ser marcado como um nome (`<nm>a</nm>`), sem que seja necessário explicitar o que denomina. Modelando o documento do ponto de vista do programador, e não do compilador, podemos chegar a um modelo mais simples.

6.3.1. Uma DTD para C++

Nossa DTD reflete em grande parte a gramática constante do padrão [ISO-C++], mas algumas simplificações foram feitas:

- Para evitar a resolução das diretivas de pré-processamento, elas foram modeladas como comandos.
- O modelo para expressões foi simplificado, como no fortran, para evitar o excesso de elementos. Como resultado, as regras de prioridade dos operadores não foram capturadas no modelo.

6.3.2. Importando C++

O programa de importação foi também criado com o auxílio do esqueleto gerado pela transformação *txl_genproto.xslt*, seguido de uma edição manual, e segue o mesmo esquema: um pré-processamento para a marcação associativa dos comentários e a tradução propriamente dita pelo programa TXL.

6.3.3. Um protótipo de tecedura para C++

Este exemplo ilustra como processar simultaneamente o par de arquivos Lux para tecer a documentação de um programa. A transformação *cpp_filesummary.xslt* recebe o documento XML que contém um fonte C++ e procura, no seu interior, por

declarações de classes[15]. O arquivo XHTML gerado possui a declaração da classe (ilustra como procurar um elemento no código), suas classes-bases (ilustra uma busca descendente a partir da declaração da classe), o espaço nominal onde foi declarada (busca ascendente), conta o número de membros (busca horizontal), e as possíveis descrições contidas no arquivo RDF correspondente (busca em outro documento). Um exemplo do resultado, usando uma folha de estilos simples *(cpp_filesummary.css)*, pode ser observado na Figura 12.

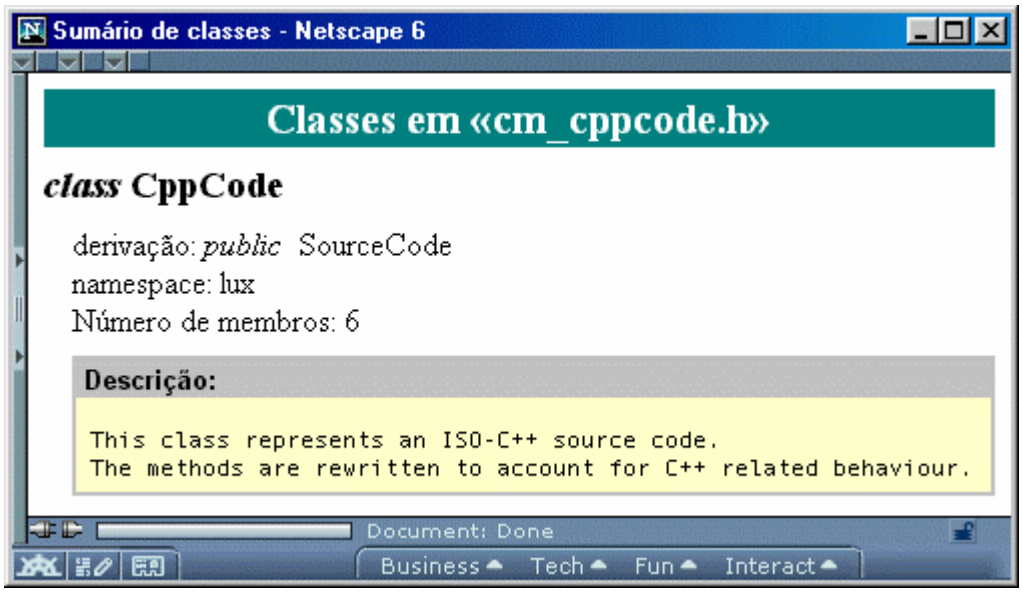


Figura 12 – Sumário de um hiperfonte C++

Uma transformação completa para tecedura será muito mais complexa, pois o código C++ é tipicamente muito distribuído: as declarações e definições estão espalhadas em diversos arquivos. Uma implementação concreta provavelmente será feita em várias etapas: a primeira, criando tabelas de símbolos para cada arquivo (as tabelas para os arquivos de inclusão do compilador podem ser pré-definidas); a segunda, criando as páginas para cada módulo de documentação (espaços nominais, classes, enumerações, etc.); e a terceira compilando sumários e índices.

---

[15] No jargão C++, classe é qualquer estrutura introduzida pelas palavras chaves *class*, *struct* ou *union*.

```
root> main
program terminated successfully
with exit code 0
root> _
```

7. CONCLUSÕES E PERSPECTIVAS

Espera-se que a separação entre conteúdo e formato que acompanha o paradigma das linguagens genéricas de marcação provocará grandes mudanças na codificação de vários tipos de documento. A migração para formatos abertos e simples, que sempre foram usados no desenvolvimento de sistemas (com algumas exceções), vem atingindo várias outras áreas, sobretudo a de processamento de texto. É razoável que em contrapartida os desenvolvedores de sistema absorvam a tecnologia já sedimentada de marcação e a utilizem para elevar a qualidade do código-fonte a um novo patamar.

Mostramos que usando uma linguagem de marcação genérica (XML), é possível armazenar código e documentação de uma forma mais versátil, universalizada e sem perda de conteúdo. Mais do que isso, esse formato facilita enormemente a manipulação automática do código, pois transformações sofisticadas podem ser escritas usando uma linguagem poderosa (XSLT) e interpretadores disponíveis na Internet a custo nulo.

Apesar das dificuldades envolvidas em manipular código em linguagens com gramáticas sensíveis ao contexto, ou muito complexas, ou ambas (como C++),

podemos usar os modelos simplificados que, mesmo com limitações, podem ser analisados, formatados e manipulados de maneiras que seriam inviáveis numa codificação textual tradicional.

Espera-se que a metodologia e os exemplos aqui contidos possam servir de base para o desenvolvimento de *frameworks* e sistemas especializados para a manipulação de código-fonte com recursos de automação e visualização indisponíveis nos atuais sistemas que usam texto simples.

7.1. Perspectivas de desenvolvimento

O uso de XML na codificação de programas-fontes implica numa forma diferente de enxergá-los. Eles deixam de ser apenas um conjunto de instruções para o compilador, e se tornam documentos que podem ser publicados, revisados e catalogados. A migração do texto simples para o marcado será tão rápida e bem sucedida na medida em que surjam ferramentas robustas para suportar as idéias aqui descritas. Seguem-se algumas sugestões para desenvolvimentos futuros.

7.1.1. Editores

Seria altamente desejável que os editores de programas pudessem editar diretamente o hiperfonte, mas a viabilidade de um editor capaz de manipular diretamente os documentos de linguagens complexas como C++ parece pequena. O uso de modelos simplificados de documento não traz maiores atrativos, pois apresentam poucas vantagens sobre os sofisticados editores hoje existentes para o fonte tradicional. Uma solução híbrida, que alterne entre as versões textual e marcada do fonte, pode viabilizar sistemas a curto prazo. A documentação, composta de fragmentos

XHTML, pode ser editada separadamente e usar *xpointers* para indicar o trecho ao qual se referem.

É interessante observar que é mais crucial (e fortuitamente, simples) criar ou adaptar editores para os comentários do que para o código-fonte propriamente dito. O programa pode ser editado convenientemente na versão enleada, e depois reimportado para o formato XML, ao passo que os comentários carecem de um editor com mais recursos, que crie e preserve *links* diversos.

7.1.2. Esquemas de documentação

Os arquivos RDF que contêm a documentação do código permitem utilizar esquemas de documentação altamente sofisticados. Tais esquemas podem incluir marcações específicas para autores, revisões, palavras-chaves, *links* para outros documentos e muito mais. Organizados sob a tutela de um gerenciador de banco de dados com suporte a RDF, é possível realizar consultas sofisticadas num repositório de código-fonte. Neste aspecto, a integração com os sistemas atuais de controle de versão é altamente desejável.

Há um grande esforço no desenvolvimento de agentes para busca inteligente de informação na Grande Rede, e RDF parece mostrou-se uma das poucas soluções viáveis. O uso de RDF para armazenar metainformação sobre o código introduz os fontes de programa na categoria prestigiada de "recurso disponível na Internet", podendo ser identificados por URIs. Isso é de especial importância para o desenvolvimento de *software* aberto, mas funciona com igual eficácia no interior das redes privadas corporativas.

7.1.3. Tradutores

O armazenamento de uma estrutura arbórea que representa a estrutura gramatical do código permite escrever tradutores em XSLT para converter código de uma linguagem de programação para outra, se as linguagens forem semelhantes. Longe de insinuar que isto é uma tarefa simples, como deixa claro [Terekhov], é possível contudo criar tradutores com escopo reduzido. Por exemplo, se um código fortran não possui comandos de entrada e saída, sua conversão para C++ é razoavelmente simples [Amaral-99].

7.1.4. Compiladores

A adaptação de compiladores para receber entrada em XML passa pela sedimentação de um modelo de documento para a linguagem de programação. Isso dependerá do sucesso desta forma de codificar, mas existem limitações técnicas; a análise é até mais simples que para o código-fonte tradicional. Em particular, compiladores escritos em XSLT podem ser utilizados para estudos exploratórios sem a necessidade de se recorrer a ferramentas sofisticadas de *parsing*.

REFERÊNCIAS BIBLIOGRÁFICAS

[Amaral-00] Amaral, Marcelo Jaccoud. *For-See: Um visualisador para código fortran*. Projeto final de programação. Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), junho de 2000.

[Amaral-99] Amaral, Marcelo Jaccoud. Traduzindo código legado sob a ótica do mantenedor: um estudo prático usando Fortran e C++. Trabalho final de Tópicos Avançados de Orientação a Objeto. Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), dezembro de 1999.

[Armstrong] Armstrong, Eric. *Encoding Source in XML: a Strategic Analysis*
http://www.treelight.com/software/encodingSource.html
http://www.treelight.com/software/encodingSample.html
http://www.treelight.com/software/XmlEditor.html

[Baecker] Baecker, Ronald; Marcus, Aaron. Printing and Publishing C Programs, in: *Software Visualization — Programming as a Multimedia Experience* (edited by Stasko, J., Domingue, J., Brown. M. H. and Price, B. A.). The MIT Press.

[Bellay] Bellay, Berndt; Gall, Harald. An Evaluation of Reverse Engineering Tool Capabilities. *J. Softw. Maint: Res. Pract.* **10**, 305-331 (1998). John Willey & Sons, Ltd.

[Braga] Braga, Christiano de Oliveira; Staa, Arndt von; Leite, Júlio César Sampaio do Prado. *Documentu*: A Flexible Architecture for Documentation Production Based on a Reverse-engineering Stategy. *J. Softw. Maint: Res. Pract.* **10**, 279-303 (1998)

[Briggs] Briggs, Preston. *Nuweb Version 0.91 — A Simple Literate Programming Tool.*
http://www-oss.fnal.gov/~mengel/nuweb/nuweb/nuweb.html

[Brown-Childs] Brown, Marcus; Childs, Bart. An Interactive Enviroment for Literate Programming. *Structured Programming* **11**:11-25 (1990), Springer-Verlag.

[Brown-Cordes] Brown, Marcus; Cordes, David. Literate Programming Applied to Conventional Software Design. *Structured Programming* **11**:85-98 (1990), Springer-Verlag

[Childs] Childs, Bart; Samentiger, Johannes. Analysis of Literate Programs from the Viewpoint of Reuse. *Software—Concepts and Tools* **18**:35-46 (1996)

[Coates] Coates, Anthony B. *XML and Literate Programming.* http://www.ems.uq.edu.au/Seminars/XML_LitProg/index.html

[Cordy] Cordy, James, R.; Carmichael, Ian, H.; Halliday, Russel. *The TXL Programming Language – Version 8.* Software Technology Laboratory, Department of Computing and Information Science, Queen's University, Canada

[Cowan] Cowan, D. D.; Germán, D. M.; Lucena, C. J. P.; Staa, A. von. *Enhancing Code for Readability and Comprehension Using SGML.* March 22, 1994

[CSF] Software Development Foundation (SDS) Project Site. *Code Structure Format (CSF) Specification*. http://sds.sourceforge.net/

[CSS2] Bos, Bert; Lie, Håkon Wium; Lilley, Chris; Jacobs, Ian. *Cascading Style Sheets, level 2 — CSS2 Specification.* W3C Recommendation, maio de 1998. http://www.w3.org/TR/REC-CSS2

[DC] Dublin Core Metadata Initiative. http://purl.org/dc

[Felix] Felix, Marcelo Fagundes. *LET: Uma linguagem para especificar traduções e seu compilador.* Dissertação de mestrado. Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 1998.

[Germán] Germán, D. M.; Cowan, D. D.; Ryman, A. *SGML-Lite — An SGML-based Programming Enviroment for Literate Programming*

[Graham] Graham, Ian S.; Quin, Liam. XML Specification Guide. John Wiley & Sons, Inc., 1999.

[ISO-C++] International Standard ISO/IEC 14882:1998(E) Programming languages — C++. American National Standards Institute, primeira edição, 1998.

[Jellife] Jelliffe, Rick. *The Schematron Assertion Language*. Academia Sinica Computing Centre. http://www.ascc.net/xml/resources/schematron/Schematron2000.html

[Kay] Kay, Michael. *XSLT Programmer's Reference.* Wrox Press, 2000.

[Knuth-83] Knuth, Donald Ervin. *The WEB system of structured documentation.* Stanford Computer Science Report CS980, Stanford University, Stanford, CA, September 1983.

[Knuth-84] Knuth, Donald Ervin. *Literate programming.* The Computer Journal, 27(2):97–111, May 1984.

[Knuth-91] TEX: The Program. *Computers & Typesetting, vol. B.* Quarta edição, Addison-Wesley, 1991.

[Knuth-92] Knuth, Donald Ervin. *Literate Programming.* Stanford University Center for the Study of Language and Information, April 1992. ISBN 0-937073-80-6

[Knuth-Levy] Knuth, Donald Ervin; Levy, Silvio. *The CWEB System of Structured Documentation.* Reading, Massachusetts: Addison-Wesley, 1993. ISBN 0-201-57569-8.

[Krommes] Krommes, John A. *FWEB: A WEB system of structured documentation for multiple languages.* http://w3.pppl.gov/~krommes/fweb_toc.html

[Leite] Leite, J. C. S. P.; Sant'Anna, M.; Freitas, F. Draco-PUC: A technology assembly for domain oriented software development. *Proceedings of the 3rd IEEE InternationalConference on Software Reuse:* 94-101 (1994). IEEE Computer Society Press.

[Lisa] Friendly, Lisa. The Design of Distributed Hyperlinked Programming Documentation. Sun Microsystems Inc. ftp://ftp.java.sun.com/docs/javadoc-paper/iwhd.pdf

[McConnell] McConnell, Steve. *Software Project Survival Guide*. Microsoft Press, 1998.

[Olsem] Olsem, Michael R. An Incremental Approach to *Software* Systems Re-engineering. *J. Softw. Maint: Res. Pract.* **10**, 181-202 (1998)

[Ramsey] Ramsey, N. Literate programming simplified. *IEEE Software* **11** (5): 97-105, 1994.

[RDF] Lassila, Ora; Swick, Ralph R. *Resource Description Framework (RDF) Model and Syntax Specification.* W3C Recommendation, fevereiro de 1999. http://www.w3c.org/TR/REC-rdf-syntax

[RDF-Schema] Brickley, Dan; Guha, R.V. *Resource Description Framework (RDF) Schema Specification 1.0.* W3C Candidate Recommendation, março de 2000. http://www.w3c.org/TR/rdf-schema

[Ream] Ream, Edward K. *LEO's Home Page.* http://www.mailbag.com/users/edream/front.html

[Ross] Williams, Ross. *FunnelWeb.* http://www.ross.net/funnelweb/

[Staa] Staa, Arndt von. *Ambiente de Engenharia de Software Talisman, Manual do Usuário*. Staa Informática, Rio de Janeiro.

[Staa-00] Staa, Arndt von. *Programação Modular.* Editora Campus. 87-108, 2000.

[Sun] Sun Microsystems. Java™ Code Conventions.
http://java.sun.com/docs/codeconv/index.html
ftp://ftp.javasoft.com/docs/codeconv/CodeConventions.pdf

[Terekhov] Terekhov, Andrey A.; Verhoef, Chris. The Realities of Language Conversions. *IEEE Softw.* Nov./Dez. 2000.

[Thimbleby] Thimbelby, H. Experiences of Literate Programming Using CWEB. *Comput. J.* **(29)** 3:201-211, 1986

[Walsh] Walsh, Norman; Muellner, Leonard. *DocBook — The Definitive Guide.* O'Reilly & Associates, 1999.

[XHTML] Pemberton, Steven; et alii. *XHTML ™ 1.0: The Extensible HyperText Markup Language. A Reformulation of HTML 4 in XML 1.0.* W3C Recommendation, janeiro de 2000
http://www.w3.org/TR/xhtml

[XML] Bray, Tim; Paoli, Jean; Sperberg-McQueen, C. M. *Extensible Markup Language (XML) 1.0.* W3C Recommendation, fevereiro de 1998.
http://www.w3.org/TR/REC-xml

[XSL-FO] Extensible Stylesheet Language (XSL). W3C Working Draft.
http://www.w3.org/TR/WD-xsl

[XSLT] Clark, James. *XSL Transformations (XSLT) Version 1.0.* W3C Recommendation, novembro de 1999.
http://www.w3.org/TR/xslt

BIBLIOGRAFIA

Boumphrey, Frank; Greer, Cassandra; Raggett, Dave; Raggett, Jenny; Schnitzenbaumer, Sebastian; Wugofski, Ted. *Beginning XHTML.* Wrox Press Ltd., April 2000.

Marchal, Benoît. *XML by Example.* Que Publishing Company, 2000.

Meyer, Eric A. *Cascading Style Sheets: The Definitive Guide.* O'Reilly & Associates, Inc, May 2000.

St. Laurent, Simon; Biggar, Robert. *Inside XML DTDs.* McGraw-Hill, 1999.

APÊNDICE

Este apêndice contém as listagens dos modelos de documento, folhas de estilo, transformações e gramáticas utilizados na pesquisa. Uma coleção completa destes arquivos, acrescidos dos fontes dos programas TXL e C++, pode ser obtida por solicitação ao autor.

txl.dtd

```
<!ENTITY % cdc-sequence  "(constructor | deconstructor | condition)*" >

<!ELEMENT program  (define | rule | external | keys | tokens | compounds | comments |
              include)* >

<!ELEMENT define  (nm,derivation*) >

<!ELEMENT nm  (#PCDATA) >

<!ELEMENT derivation  (lit | type)* >

<!ELEMENT lit  (#PCDATA) >

<!ELEMENT type  (nm | lit) >

<!ELEMENT rule  (nm,arg*,%cdc-sequence;,skipping?,((replace,%cdc-sequence;,by) | (
              match,%cdc-sequence;))) >

<!ELEMENT arg  (nm,type) >

<!ELEMENT constructor  (nm,type,replacement) >

<!ELEMENT replacement  (lit | expression)* >

<!ELEMENT expression  (nm,application*) >

<!ELEMENT application  (nm,(nm | lit)*,each?) >

<!ELEMENT each  (nm*) >

<!ELEMENT deconstructor  (type?,nm,pattern) >

<!ELEMENT pattern  ((nm,type?) | lit)* >

<!ELEMENT condition  (expression) >

<!ELEMENT skipping  (type) >

<!ELEMENT replace  (type,pattern) >

<!ELEMENT by  (replacement) >

<!ELEMENT match  (type,pattern) >

<!ELEMENT external  (nm,arg*) >

<!ELEMENT keys  (lit | nm | kw)* >

<!ELEMENT kw  (#PCDATA) >

<!ELEMENT tokens  (token-def*) >

<!ELEMENT token-def  (nm?,plus?,lit) >

<!ELEMENT plus   EMPTY  >

<!ELEMENT compounds  (lit | nm | kw)* >

<!ELEMENT comments  (convention*) >

<!ELEMENT convention  (lit,lit?) >

<!ELEMENT include  (lit) >

<!ATTLIST application
              mod  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST arg
              id  ID    #IMPLIED  >

<!ATTLIST by
```

```
              id  ID    #IMPLIED  >

<!ATTLIST comments
              id  ID    #IMPLIED  >

<!ATTLIST compounds
              id  ID    #IMPLIED  >

<!ATTLIST condition
              mod  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST constructor
              id  ID    #IMPLIED  >

<!ATTLIST convention
              id  ID    #IMPLIED  >

<!ATTLIST deconstructor
              id  ID    #IMPLIED
              apply (search | once | repeat)   #IMPLIED  >

<!ATTLIST define
              id  ID    #IMPLIED  >

<!ATTLIST derivation
              id  ID    #IMPLIED  >

<!ATTLIST expression
              id  ID    #IMPLIED  >

<!ATTLIST external
              id  ID    #IMPLIED
              apply (search | once | repeat)   #IMPLIED  >

<!ATTLIST include
              id  ID    #IMPLIED  >

<!ATTLIST keys
              id  ID    #IMPLIED  >

<!ATTLIST lit
              t (string | char | number | quote | free)   #REQUIRED
              xml:space (preserve | default)  "default" >

<!ATTLIST match
              id  ID    #IMPLIED  >

<!ATTLIST pattern
              id  ID    #IMPLIED  >

<!ATTLIST program
              id  ID    #IMPLIED  >

<!ATTLIST replace
              id  ID    #IMPLIED  >

<!ATTLIST replacement
              id  ID    #IMPLIED  >

<!ATTLIST rule
              id  ID    #IMPLIED
              apply (search | once | repeat)   #IMPLIED  >

<!ATTLIST skipping
              id  ID    #IMPLIED  >

<!ATTLIST token-def
              id  ID    #IMPLIED  >

<!ATTLIST tokens
              id  ID    #IMPLIED  >

<!ATTLIST type
              mod  CDATA    #IMPLIED  >
```

txl.css

```
/*
   PUC-Rio - Departamento de Informática

   Arquivo:    txl.css
   Conteúdo:   Folha de estilos para documentos do tipo definido por txl.dtd
   Autor:      Marcelo Jaccoud Amaral
   Data:       02/05/2001

*/


/*
   Estes elementos são particulares do XMetaL, e correspondem aos
   comentários e instruções de processamento XML. Estas estruturas não
   têm suporte direto em CSS2.
*/


$COMMENT {
  display: block;
  color: purple;
  white-space: pre;
}

$PROCINS {
  color: black;
  background-color: #c0c0c0;
}


/* Elementos comuns */

/*
   Estilos gerais, aplicados ao elementos mais externo. O efeito em
   cascata aplica-os a quase todos os elementos.
*/
program {
  display: block;
  font-family: Arial, Helvetica, "Sans Serif", Serif, proportional;
  font-size: 11pt;
  margin-top: 5px;
  margin-left: 5px;
  font-style: normal;
  font-weight: normal;
  background-color: #FFFFFF;
  display: block;
  text-align: left;
}


/* operadores: garante que seja delimitado visualmente por espaços. */
op {
  font-weight: bold;
}
op:before, op:after {
  content: " ";
}


/* os tipos e aplicações são distinguidos pela cor dos colchetes */
type:before {
  content: "[ ";
  display: inline;
  color: #66BBCC;
  font-weight: bold;
}

type:after {
  content: "] ";
  display: inline;
  color: #66BBCC;
  font-weight: bold;
```

```
}

application:before {
  content: "[ ";
  display: inline;
  color: #00CC00;
  font-weight: bold;
}

application:after {
  content: "] ";
  display: inline;
  color: #00CC00;
  font-weight: bold;
}

/* reforça os nomes dos defines e das regras de transformação */
define>nm, rule>nm {
  font-weight: bold;
}

nm {
  margin-right: 1pt;
  margin-left: 1pt;
  color: #6666CC;
  display: inline;
}


/*
   Estilos para os nomes e literais
*/

nm:after {
  content: " ";
}

compounds>lit, keys>lit {
  display: block;
  margin-left: 2em;
}

lit[t="string"] {
  display: inline;
  font-weight: normal;
  font-family: monospace;
}

lit[t="quote"] {
  font-weight: bold;
  color: #0000AA;
}

lit[t="quote"]:before {
  content: " '";
  font-weight: bold;
  color: #0000AA;
}

lit[t="quote"]:after {
  content: " ";
}

lit[t="string"]:before {
  content: " «";
}

lit[t="string"]:after {
  content: "» ";
}

/*
   Estilos para o leiaute de cada seção do programa
*/
```

```
define, keys, rule, tokens, compounds, comments, external, include {
  margin-top: 12pt;
  margin-bottom: 12pt;
  display: block;
  background-color: #FFFEE3;
  border: 2pt #FFCC99 solid;
}

by, match, condition, replacement, pattern, arg, constructor, deconstructor, skipping,
convention, replace {
  display: block;
  margin-left: 2em;
}

comments:before, comments:after, keys:before, compounds:before, keys:after, compounds:after
{
  display: block;
  color: #FF6600;
}

condition:before, define:before, rule:before, tokens:before, tokens:after, external:before,
include:before, define:after, rule:after, deconstructor:before, constructor:before,
match:before, by:before, replace:before, skipping:before {
  display: inline;
  color: #FF6600;
}

condition[mod="not"]:before {
  content: "where not ";
}

condition[mod="all"]:before {
  content: "where all ";
}

condition[mod="notall"]:before {
  content: "where not all ";
}

condition:before {
  content: "where ";
}

comments:before {
  content: "comments ";
}

comments:after {
  content: "end comments ";
}

keys:before {
  content: "keys ";
}

keys:after {
  content: "end keys ";
}

define:before {
  content: "define ";
}

define:after {
  content: "end define ";
}

rule[apply="repeat"]:before {
  content: "rule ";
}

rule[apply="repeat"]:after {
  content: "end rule ";
}

rule[apply="search"]:before {
```

```
  content: "function (*) ";
}

rule:before {
  content: "function ";
}

rule:after {
  content: "end function ";
}

external:before {
  content: "external ";
}

tokens:before {
  content: "tokens ";
  display: block;
}

tokens:after {
  content: "end tokens ";
  display: block;
}

compounds:before {
  content: "compounds ";
}

compounds:after {
  content: "end compounds ";
}

include:before {
  content: "include ";
}

skipping:before {
  content: "skipping ";
}

rule[apply='search'] replace:before {
  content: "replace * ";
}

replace:before {
  content: "replace ";
}

by:before {
  content: "by ";
}

match:before {
  content: "match ";
}

constructor:before {
  content: "construct ";
}

deconstructor:before {
  content: "deconstruct ";
}

appplication, type {
  display: inline;
}

define {
  counter-reset: deriv-count;
}

derivation {
  counter-increment: deriv-count;
  margin-left: 5em;
```

```
  display: block;
  text-indent: -4em;
}

derivation:before {
  content: "(" counter(deriv-count, lower-alpha) ") ";
  font-family: monospace;
  font-size: 11pt;
  color: #FF6600;
  font-style: italic;
}

function, rule {
  counter-reset: arg-count;
}

arg {
  counter-increment: arg-count;
}

arg:before {
  content: "(" counter(arg-count) ") ";
  font-family: monospace;
  font-size: 11pt;
  color: #FF6600;
  font-style: italic;
}


pattern {
  background-color: #E9FFFE;
}

replacement {
  background-color: #EAFFE9;
}

tokens {
  display: block;
}

token-def {
  display: block;
  margin-left: 4em;
  text-indent: -2em;
}

plus {
  display: none;
}

plus:before {
  content: "+";
  border: 1pt solid;
  color: #6666CC;
  font-weight: bold;
}

/* fim de txl.css */
```

rdf.css

```
/*
   PUC-Rio - Departamento de Informática

   Arquivo:    rdf.css
   Conteúdo:   Folha de estilos para os documentos RDF do sistema Lux.
   Autor:      Marcelo Jaccoud Amaral
   Data:    02/05/2001

*/

/*
   Atenção:
   ========

   CSS2 não tem suporte aos espaços nominais (namespaces), que são
   fundamentais em RDF, pois distinguem os elementos dos diversos tipos de
   documento referenciados. Esta folha de estilos pode não ter o efeito
   desejado em alguns processadores. Ela foi testada e funciona corretamente
   no XMetaL 2.1 e no Netscape Navigator 6.01.
*/


/* Em verdade rdf:Description, onde rdf é o espaço nominal do padrão RDF */
Description {
  display: block;
  margin-top: 4pt;
  margin-bottom: 8pt;
  margin-left: 4pt;
  margin-right: 4pt;
  border: 1.5pt silver solid;
  font-size: 11pt;
  font-family: Arial;
}

Description:before {
  content: "Descrição de " attr(about);
  display: block;
  padding-left: 2pt;
  font-weight: bold;
  background-color: silver;
  border: 1pt silver solid;
}

/* estes elementos estão no espaço nominal do Dublin Core (dc) */
format, language, source, type {
  display: block;
  background-color: #DEFFDD;
  padding-left: 20pt;
}

format:before {
  content: "formato: ";
}

language:before {
  content: "linguagem: ";
}

source:before {
  content: "arquivo-fonte: ";
}

type:before {
  content: "tipo: ";
}

format:before, language:before, source:before, type:before {
  font-weight: bold;
}

/* Este elemento está no espaço nominal Lux (lux) */
description {
```

```
  font-family: "Andale Mono";
  font-size: 80%;
  background-color: #FFFFCC;
  display: block;
  white-space: pre;
  padding: 2pt;
}

/* fim de rdf.css */
```

txl.grm

```
% TXL 7.0 language grammar
% J.R. Cordy, Queen's University, August 1992
% Copyright 1995 Legasys Corp.

% Customized for use with txlpara -- JRC 25.8.1993
% Updated to handle TXL 8.0 -- JRC 5.9.1995
% Adapted to enable full TXL translation into XML
%  -- Marcelo Jaccoud Amaral 10.3.2000

% this pragma sets the txl and comment run-time options to allow
% a TXL program to process its own sintax
%# pragma -txl -comment

% TXL comments begin with a '%' and end at end of line.
% The '%' character must be quoted if it appears as
% part of a TXL program.

comments
   '%
end comments


% The following are keywords of TXL and must be quoted if they
% appear in a TXL program with other than their TXL meaning.
% This grammar is such a program, since it is TXL code meant to
% handle TXL code.

keys
   'all           'attr          'by            'comments
   'compounds     'construct     'deconstruct   'define
   'each          'end           'external      'function
   'include       'keys          'list          'match
   'not           'opt           'repeat        'replace
   'rule          'skipping      'tokens        'where
end keys


% The id extension allows for the anonymous variable used in rules.
% The digraph token is used to parse literals. Note that this
% definition is incomplete, since it does not allow certain types of
% unquoted character sequences. A failproof definition would include the
% comma symbol, but that produces a problem parsing the type suffix ',+'
% which I could not trace [mja].

tokens
   id + "_"
   digraph "[|\#<>:.\+-\*/!^&~=\(\){}\?][|\#<>:.\+-\*/!^&~=\(\){}\?;]+"
   digraph + ".\i+."
end tokens


% This is the starting symbol

define program
   [statement*]
end define


% The TXL Statements

define statement
   [define_statement]          [NL][NL]
    | [rule_statement]         [NL][NL]
    | [function_statement]     [NL][NL]
    | [external_statement]     [NL][NL]
    | [keys_statement]         [NL][NL]
    | [tokens_statement]       [NL][NL]
    | [compounds_statement]    [NL][NL]
    | [comments_statement]     [NL][NL]
    | [include_statement]      [NL][NL]
    | [comment_block]          [NL][NL]
end define
```

```
% Comment block between statements

define comment_block
   [comment+]
end define


% include statement

define include_statement
   'include [stringlit]
end define


% keys statement

define keys_statement
   'keys                    [NL][IN]
      [literals]            [NL][EX]
   'end 'keys               [NL]
end define


% compounds statement. To parse this correctly, some literals actually
% allowed were left out, namely those containing weird mixtures of letters
% and special characters. This should be however healthy, since a sintax with
% such strange literals will probably not work correctly.

define compounds_statement
   'compounds               [NL][IN]
       [literals]           [NL][EX]
   'end 'compounds          [NL]
end define


% The comments statement cannot be parsed accurately by TXL, because its
% syntax relies on newline characteres, which are ignored by the parser.
% To allow this section to be correctly parsed, use comments at the and of
% each convention line (empty comments are OK).

define comments_statement
   'comments                [NL][IN]
      [comment_block?]
      [comment_entry*]      [NL][EX]
   'end 'comments           [NL]
end define

define comment_entry
   [comment_convention] [comment_block?] [NL]
end define

define comment_convention
   [bounded_comment_convention]
   |
   [eol_comment_convention]
end define

define eol_comment_convention
   [literal]
end define

define bounded_comment_convention
   [literal][literal]
end define


% tokens statement

define tokens_statement
   'tokens                  [NL][IN]
      [comment_block?]
      [token_entry*]        [EX]
   'end 'tokens             [NL]
end define
```

```
define token_entry
   [token_definition] [comment_block?] [NL]
end define

define token_definition
   [tokenid] [stringlit]                [NL]
   | [tokenid] '+ [stringlit]           [NL]
   | [IN][IN] '+ [stringlit] [EX][EX][NL]
   | [comment]
end define


% define statement

define define_statement
   'define [typeid]             [comment_block?]     [NL][IN]
       [derivation*]
       [bar_derivation*]                             [EX][NL]
       [comment_block?]
   'end 'define                                      [NL]
end define

define derivation
   [literal_or_type] [comment_block?]
end define

define literal_or_type
   [literal] | [type]
end define

define bar_derivation
    [NL] '| [comment_block?] [derivation*]
end define


% rule statement

define rule_statement
   'rule [ruleid] [comment_block?] [formal_argument*] [NL]
   [repeat construct_deconstruct_or_condition]
   [opt skipping_type]
   [IN] 'replace [type] [comment_block?] [NL]
   [IN] [pattern] [EX][EX][NL]
   [repeat construct_deconstruct_or_condition]
   [IN] 'by [comment_block?] [NL]
   [IN] [replacement] [EX][EX][NL]
   'end 'rule
   |
   'rule [ruleid] [comment_block?] [repeat formal_argument] [NL]
   [repeat construct_deconstruct_or_condition]
   [opt skipping_type]
   [IN] 'match [type] [comment_block?] [NL]
   [IN] [pattern] [EX][EX][NL]
   [repeat construct_deconstruct_or_condition]
   'end 'rule
end define


% function statement

define function_statement
   'function [ruleid] [comment_block?] [repeat formal_argument] [NL]
   [repeat construct_deconstruct_or_condition]
   [opt skipping_type]
   [IN] 'replace [opt '*] [type] [comment_block?] [NL]
   [IN] [pattern] [EX][EX][NL]
   [repeat construct_deconstruct_or_condition]
   [IN] 'by [comment_block?] [NL]
   [IN] [replacement] [EX][EX][NL]
   'end 'function
   |
   'function [ruleid] [comment_block?] [repeat formal_argument] [NL]
   [repeat construct_deconstruct_or_condition]
   [opt skipping_type]
   [IN] 'match [opt '*] [type] [comment_block?] [NL]
```

```
   [IN] [pattern] [EX][EX][NL]
   [repeat construct_deconstruct_or_condition]
   'end 'function
end define


% external statement

define external_statement
   'external 'rule [ruleid] [repeat formal_argument]
   | 'external 'function [ruleid] [repeat formal_argument]
end define


% common non-terminals used by more than one statement

% type definitions

define type
   [SPOFF] '[ [typeid] '] [SPON]
   | [SPOFF] '[ 'opt    [SP][id_or_quoted_literal]                 '] [SPON]
   | [SPOFF] '[ 'repeat [SP][id_or_quoted_literal] [plus_or_star?] '] [SPON]
   | [SPOFF] '[ 'list   [SP][id_or_quoted_literal] [plus_or_star?] '] [SPON]
   % Alternative short forms for the above
   | [SPOFF] '[ [id_or_quoted_literal] [type_suffix] '] [SPON]
end define

define id_or_quoted_literal
   [typeid]
   | [quoted_literal]
end define

define plus_or_star
   '+ | '*
end define

define type_suffix
   '? | '* | '+ | ', | ',+
end define

% rule/function sections

define formal_argument
   [varid] [type] [comment_block?]
end define

define construct_deconstruct_or_condition
   [constructor]
   | [deconstructor]
   | [condition]
end define

define constructor
   [IN] 'construct [varid] [type] [comment_block?] [NL]
   [IN] [replacement] [EX][EX][NL]
end define

define deconstructor
   [IN] 'deconstruct [opt '*] [opt type] [varid] [comment_block?] [NL]
   [IN] [pattern] [EX][EX][NL]
end define

define condition
   [IN] 'where [opt 'not] [opt 'all] [comment_block?] [NL]
   [IN] [expression] [EX][EX][NL]
end define

define skipping_type
   [IN] 'skipping [type] [comment_block?] [EX][NL]
end define

define pattern
   [repeat literal_or_variable]
end define

define literal_or_variable
```

```
   [literal] [comment_block?]
   | [new_variable] [comment_block?] % introduction of a new variable
   | [old_variable] [comment_block?] % reference to a previously defined variable
end define

define new_variable
   [varid] [type]
end define

define old_variable
   [varid]
end define

define replacement
   [repeat literal_or_expression]
end define

define literal_or_expression
   [literal] [comment_block?]
   | [expression]
end define

define expression
   [varid] [comment_block?] [repeat rule_application]
end define

define rule_application
   '[ [opt '?] [ruleid] [repeat varid_or_literal] [opt 'each] [repeat varid]
   '] [comment_block?]
end define

define varid_or_literal
   [varid] | [literal]
end define

% literals

define literals
   [literal_decl+]
   [comment_block?]
end define

define literal_decl
   [comment_block?]
   [spec_literal]
end define

define spec_literal
   [id]
   | [digraph]
   | [quoted_literal]
   | [special_seq]
end define

define quoted_literal
   ' '  [quoted_token]
end define

define special_seq
   [special] [digraph]
   | [special+]
end define

define quoted_token
   [key]
   | [id]
   | [digraph]
   | [special]
   | '|
   | '[
   | ']
   | ' '
end define

define literal
   [quoted_literal]
```

```
    | [stringlit]
    | [charlit]
    | [number]
    | [special]
    | [digraph]
end define


define special
      '! | '@ | '# | '$ | '^ | '& | '* | '( | ')
    | '_ | '+ | '{ | '} | ': | '< | '> | '? | '~
    | '\ | '= | '- | '; | ', | '. | '/ | '%
end define


% identifiers

define varid
   [id]
end define

define typeid
   [id]
end define

define ruleid
   [id] | [built_in_rule]
end define

define tokenid
   [id]
end define

define built_in_rule
   '. | '_ | '# | ': | '! | ', | '^ | '$
      | '+ | '- | '* | '/
      | '= | '~= | '> | '>= | '< | '<=
end define

% end of txl.grm
```

txl_tangle.xslt

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--
   PUC-Rio - Departamento de Informática

   Arquivo:    txl_tangle.xslt
   Projeto:    Lux
   Conteúdo:   Transformação para enleio de código TXL.
   Autor:      Marcelo Jaccoud Amaral
   Data:       2/5/2001

   Descrição:
      Esta transformação gera um arquivo texto codificado
      em ISO-9959-1 (Latin-1) com o código-fonte TXL
      equivalente ao documento de entrada, que deve ser
      do tipo txl.dtd.
      Esta operação é conhecida por enleio (ingl. tangle).
-->

<xsl:stylesheet
   version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >

<xsl:output method="text" encoding="ISO-8859-1"/>

<!-- Gabaritos auxiliares -->

<xsl:template name="insert-label">
   <xsl:if test="@id">
      <xsl:text>  % id(</xsl:text>
      <xsl:value-of select="@id"/>
      <xsl:text>)</xsl:text>
   </xsl:if>
   <xsl:text>&#x0a;</xsl:text>
</xsl:template>

<xsl:template name="unique-label">
   <xsl:if test="not(.//*[@id]) and @id">
      <xsl:text>  % id(</xsl:text>
      <xsl:value-of select="@id"/>
      <xsl:text>)&#x0a;</xsl:text>
   </xsl:if>
</xsl:template>

<xsl:template name="space-before">
   <xsl:choose>
      <xsl:when test="parent::replacement | parent::expression">
         <xsl:text>      </xsl:text>
      </xsl:when>
      <xsl:when test="position() > 2">
         <xsl:text> </xsl:text>
      </xsl:when>
   </xsl:choose>
</xsl:template>

<xsl:template name="space-after">
   <xsl:if test="parent::replacement | parent::expression">
      <xsl:text>&#x0a;</xsl:text>
   </xsl:if>
</xsl:template>

<!-- Raiz -->

<xsl:template match="program">
%
% Este arquivo foi gerado pela stylesheet 'txl_tangle.xslt'
%

<xsl:apply-templates/>
</xsl:template>

<!--
   Elementos principais
```

```
-->

<xsl:template match="define">
   <xsl:text>define </xsl:text>
   <xsl:apply-templates select="nm"/>
   <xsl:call-template name="insert-label"/>
   <xsl:apply-templates select="derivation"/>
   <xsl:text>end define&#x0a;&#x0a;</xsl:text>
</xsl:template>

<xsl:template match="rule">
   <xsl:choose>
      <xsl:when test="@apply = 'repeat'">
         <xsl:text>rule </xsl:text>
      </xsl:when>
      <xsl:otherwise>
         <xsl:text>function </xsl:text>
      </xsl:otherwise>
   </xsl:choose>
   <xsl:apply-templates select="nm"/>
   <xsl:call-template name="insert-label"/>
   <xsl:apply-templates select="*[not(self::nm)]"/>
   <xsl:choose>
      <xsl:when test="@apply = 'repeat'">
         <xsl:text>end rule&#x0a;&#x0a;</xsl:text>
      </xsl:when>
      <xsl:otherwise>
         <xsl:text>end function&#x0a;&#x0a;</xsl:text>
      </xsl:otherwise>
   </xsl:choose>
</xsl:template>

<xsl:template match="external">
   <xsl:text>external </xsl:text>
   <xsl:choose>
      <xsl:when test="@apply = 'repeat'">
         <xsl:text>rule </xsl:text>
      </xsl:when>
      <xsl:otherwise>
         <xsl:text>function </xsl:text>
      </xsl:otherwise>
   </xsl:choose>
   <xsl:apply-templates/>
   <xsl:call-template name="insert-label"/>
   <xsl:text>end define&#x0a;&#x0a;</xsl:text>
</xsl:template>

<xsl:template match="keys">
   <xsl:text>keys </xsl:text>
   <xsl:call-template name="insert-label"/>
   <xsl:for-each select="*">
      <xsl:if test="position() mod 5 = 1">
         <xsl:text>   </xsl:text>
      </xsl:if>
      <xsl:apply-templates select="current()"/>
      <xsl:if test="position() mod 5 = 0 or position() = last()">
         <xsl:text>&#x0a;</xsl:text>
      </xsl:if>
   </xsl:for-each>
   <xsl:text>end keys&#x0a;&#x0a;</xsl:text>
</xsl:template>

<xsl:template match="tokens">
   <xsl:text>tokens </xsl:text>
   <xsl:call-template name="insert-label"/>
   <xsl:apply-templates/>
   <xsl:text>end tokens&#x0a;&#x0a;</xsl:text>
</xsl:template>

<xsl:template match="compounds">
   <xsl:text>compounds </xsl:text>
   <xsl:call-template name="insert-label"/>
   <xsl:apply-templates/>
   <xsl:text>end compounds&#x0a;&#x0a;</xsl:text>
</xsl:template>
```

```
<xsl:template match="comments">
   <xsl:text>comments </xsl:text>
   <xsl:call-template name="insert-label"/>
   <xsl:apply-templates/>
   <xsl:text>end comments&#x0a;&#x0a;</xsl:text>
</xsl:template>

<xsl:template match="include">
   <xsl:text>include </xsl:text>
   <xsl:apply-templates/>
   <xsl:call-template name="insert-label"/>
   <xsl:text>&#x0a;&#x0a;</xsl:text>
</xsl:template>

<!--
   Demais elementos não terminais
-->

<xsl:template match="derivation">
   <xsl:text>   </xsl:text>
   <xsl:if test="position() != 1">| </xsl:if>
   <xsl:apply-templates/>
   <xsl:call-template name="insert-label"/>
</xsl:template>

<xsl:template match="type">
   <xsl:call-template name="space-before"/>
   <xsl:text>[</xsl:text>
   <xsl:apply-templates/>
   <xsl:value-of select="@mod"/>
   <xsl:text>]</xsl:text>
</xsl:template>

<xsl:template match="arg">
   <xsl:text>      </xsl:text>
   <xsl:apply-templates/>
   <xsl:call-template name="insert-label"/>
</xsl:template>

<xsl:template match="constructor">
   <xsl:text>   construct </xsl:text>
   <xsl:apply-templates select="nm"/>
   <xsl:text> </xsl:text>
   <xsl:apply-templates select="type"/>
   <xsl:call-template name="insert-label"/>
   <xsl:apply-templates select="replacement"/>
</xsl:template>

<xsl:template match="deconstructor">
   <xsl:text>   deconstruct </xsl:text>
   <xsl:apply-templates select="type"/>
   <xsl:text> </xsl:text>
   <xsl:apply-templates select="nm"/>
   <xsl:call-template name="insert-label"/>
   <xsl:apply-templates select="pattern"/>
</xsl:template>

<xsl:template match="condition">
   <xsl:text>   where </xsl:text>
   <xsl:if test="contains(@mod, 'not')">
      <xsl:text>not </xsl:text>
   </xsl:if>
   <xsl:if test="contains(@mod, 'all')">
      <xsl:text>all </xsl:text>
   </xsl:if>
   <xsl:call-template name="insert-label"/>
   <xsl:apply-templates/>
</xsl:template>

<xsl:template match="skipping">
   <xsl:text>   skipping </xsl:text>
   <xsl:apply-templates/>
   <xsl:call-template name="insert-label"/>
</xsl:template>

<xsl:template match="replace | match">
```

```
   <xsl:text>   </xsl:text>
   <xsl:value-of select="local-name()"/>
   <xsl:text> </xsl:text>
   <xsl:apply-templates select="type"/>
   <xsl:call-template name="insert-label"/>
   <xsl:apply-templates select="pattern"/>
</xsl:template>

<xsl:template match="by">
   <xsl:text>   by</xsl:text>
   <xsl:call-template name="insert-label"/>
   <xsl:apply-templates/>
</xsl:template>

<xsl:template match="replacement | expression">
   <xsl:apply-templates/>
   <xsl:call-template name="unique-label"/>
</xsl:template>

<xsl:template match="application">
   <xsl:if test="position() = 2">
      <xsl:text>&#x0a;</xsl:text>
   </xsl:if>
   <xsl:text>      [</xsl:text>
   <xsl:if test="@mod">
      <xsl:text>?</xsl:text>
   </xsl:if>
   <xsl:apply-templates/>
   <xsl:text>]</xsl:text>
   <xsl:call-template name="insert-label"/>
</xsl:template>

<xsl:template match="each">
   <xsl:text> each </xsl:text>
   <xsl:apply-templates/>
</xsl:template>

<xsl:template match="pattern | token_def">
   <xsl:text>      </xsl:text>
   <xsl:apply-templates/>
   <xsl:call-template name="insert-label"/>
</xsl:template>

<xsl:template match="convention">
   <xsl:text>      </xsl:text>
   <xsl:apply-templates/>
   <xsl:call-template name="insert-label"/>
</xsl:template>

<!--
   Terminais (elementos que contêm texto)
-->
<xsl:template match="nm">
   <xsl:call-template name="space-before"/>
   <xsl:value-of select="."/>
   <xsl:call-template name="space-after"/>
</xsl:template>

<xsl:template match="lit[@t='char']">
   <xsl:call-template name="space-before"/>
   <xsl:text>&apos;</xsl:text>
   <xsl:value-of select="."/>
   <xsl:text>&apos; </xsl:text>
   <xsl:call-template name="space-after"/>
</xsl:template>

<xsl:template match="lit[@t='number']">
   <xsl:call-template name="space-before"/>
   <xsl:value-of select="."/>
   <xsl:call-template name="space-after"/>
</xsl:template>

<xsl:template match="lit[@t='quote']">
   <xsl:call-template name="space-before"/>
   <xsl:text>&apos;</xsl:text>
   <xsl:value-of select="."/>
```

```
   <xsl:text> </xsl:text>
   <xsl:call-template name="space-after"/>
</xsl:template>

<xsl:template match="lit[@t='string']">
   <xsl:call-template name="space-before"/>
   <xsl:text>&quot;</xsl:text>
   <xsl:value-of select="."/>
   <xsl:text>&quot;</xsl:text>
   <xsl:call-template name="space-after"/>
</xsl:template>

<xsl:template match="plus">
   <xsl:call-template name="space-before"/>
   <xsl:text>+</xsl:text>
</xsl:template>

</xsl:stylesheet>
<!-- fim de txl_tangle.xslt -->
```

txl_gentree.xslt

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--
   PUC-Rio - Departamento de Informática

   Arquivo:    txl_gentree.xslt
   Projeto:    Lux
   Conteúdo:   Gerador de gerador de árvores.
   Autor:      Marcelo Jaccoud Amaral
   Data:       2/5/2001

   Descrição:
      Esta transformação gera um programa completo em TXL, cujo objetivo é
      fazer um "dumping" da árvore gramatical num arquivo XML.
      Durante a depuração de uma gramática TXL, às vezes o parsing ocorre mas
      gera árvores diferentes das esperadas. O programa gerado aqui cria um
      arquivo XML com a exata estrutura hierárquica da árvore sintática gerada
      pelo parser do TXL. Abriindo tal arquivo num editor XMl ou mesmo num
      navegador é possível analisar a árvore mais facilmente que no depurador
      TXL.
      Esta transformação demonstra como gerar programas completos usando
      dados em XML (neste caso, um adefinição de gramática).
      A estrutura do gerador é muito semelhante à de txl_genproto.xslt.
-->


<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >
<xsl:param name="grammar" select="'xyz'"/>

<xsl:output method="xml" encoding="ISO-8859-1" doctype-system="txl.dtd" indent="yes"/>

<!-- Gabaritos auxiliares -->
<!-- Raiz -->

<!--=========================================================================================-->

<!-- Constantes auxiliares -->

<xsl:variable name="replace-clause">
   <replace>
      <type mod="*"><nm>token</nm></type>
      <pattern><nm>output</nm><type mod="*"><nm>token</nm></type></pattern>
   </replace>
</xsl:variable>

<!--=========================================================================================-->

<!--
   Gabarito principal, processa <program></program>, o elemento-raiz do documento,
   no estilo 'pull'. Constrói a parte comum a todos os programas tradutores e depois
   cria funções de tradução para cada elemento definido na gramática.
-->
<xsl:template match="program">
   <program>
      <!-- cria a parte comum do programa -->
      <xsl:call-template name="write-common-part"/>
      <!-- cria as funções de tradução para todos os tipos simples -->
      <xsl:apply-templates select="define"/>
      <!-- cria as funções de tradução para os tipos com 'opt' -->
      <xsl:call-template name="write-opt-types"/>
      <!-- cria as funções de tradução para os tokens -->
      <xsl:call-template name="write-token-types"/>
      <!-- cria as funções de tradução para os tipos pré-definidos -->
      <xsl:call-template name="write-predef-types"/>
   </program>
</xsl:template>

<!--=========================================================================================-->

<!--
   Gera a parte comum a todos os programas de tradução.
-->
<xsl:template name="write-common-part">
```

```
<!-- Estas declarações de inclusão se repetem em todos os programas de tradução -->
<xsl:comment>Declarações de inclusão</xsl:comment>
<include><lit t="string">lib\TxlExtrn</lit></include>
<include><lit t="string">lib\System</lit></include>
<include><lit t="string">lux_fg.txl</lit></include>
<include><lit t="string">lux_strings.txl</lit></include>
<include><lit t="string">lux_common.txl</lit></include>
<include><lit t="string"><xsl:value-of select="$grammar"/>.grm</lit></include>

<!--
   Define a função mainRule, obrigatória em todo programa TXL.
   Esta lê o nome do arquivo de uma variável do sistema e invoca a função que o traduz.
-->

<xsl:comment>Função principal (mainRule)</xsl:comment>
<rule apply="once">
   <nm>mainRule</nm>

   <replace>
      <type><nm>program</nm></type>
      <pattern><nm>p</nm><type><nm>program</nm></type></pattern>
   </replace>

   <constructor>
      <nm>enviroment</nm>
      <type><nm>stringlit</nm></type>
      <replacement>
         <expression>
            <nm>_</nm>
            <application>
               <nm>pipe</nm>
               <lit t="string">%lux%\\getenv LUX_FILE</lit>
            </application>
         </expression>
      </replacement>
   </constructor>

   <constructor>
      <nm>file_name</nm>
      <type><nm>stringlit</nm></type>
      <replacement>
         <expression>
            <nm>_</nm>
            <application>
               <nm>parse</nm>
               <nm>enviroment</nm>
            </application>
         </expression>
      </replacement>
   </constructor>

   <constructor>
      <nm>log</nm>
      <type mod="*"><nm>token</nm></type>
      <replacement>
         <expression>
            <nm>_</nm>
            <application>
               <nm>translate</nm>
               <nm>file_name</nm>
               <nm>p</nm>
            </application>
            <application>
               <nm>print</nm>
            </application>
         </expression>
      </replacement>
   </constructor>

   <by><replacement/></by>

</rule>

<!--
   Define a função translate, que cria os arquivos e dispara a cadeia de tradução
-->
```

```
<xsl:comment>Função que dispara a tradução (translate)</xsl:comment>
<rule apply="once">
   <nm>translate</nm>
   <arg><nm>file_name</nm><type><nm>stringlit</nm></type></arg>
   <arg><nm>p</nm><type><nm>program</nm></type></arg>

   <replace>
      <type mod="*"><nm>token</nm></type>
      <pattern>
         <nm>scope</nm>
         <type mod="*"><nm>token</nm></type>
      </pattern>
   </replace>

   <constructor>
      <nm>command</nm>
      <type><nm>stringlit</nm></type>
      <replacement>
         <expression>
            <nm>_</nm>
            <application>
               <nm>+</nm>
               <lit t="string">%lux%\\systime &gt;buffer.$</lit>
            </application>
         </expression>
      </replacement>
   </constructor>

   <constructor>
      <nm>before</nm>
      <type><nm>number</nm></type>
      <replacement>
         <expression>
            <nm>_</nm>
            <application>
               <nm>system</nm>
               <nm>command</nm>
            </application>
            <application>
               <nm>read</nm>
               <lit t="string">buffer.$</lit>
            </application>
         </expression>
      </replacement>
   </constructor>

   <constructor>
      <nm>xml_name</nm>
      <type><nm>stringlit</nm></type>
      <replacement>
         <expression>
            <nm>_</nm>
            <application>
               <nm>+</nm>
               <nm>file_name</nm>
            </application>
            <application>
               <nm>+</nm>
               <lit t="string">.tree.xml</lit>
            </application>
         </expression>
      </replacement>
   </constructor>

   <constructor>
      <nm>workspace</nm>
      <type mod="*"><nm>token</nm></type>
      <replacement>
         <expression>
            <nm>_</nm>
            <application>
               <nm>initiate_stream</nm>
               <nm>xml_name</nm>
            </application>
            <application>
```

```
               <nm>lux_xmldecl</nm>
            </application>
            <application>
               <nm><xsl:value-of select="$grammar"/>_program</nm>
               <nm>p</nm>
            </application>
            <application>
               <nm>close_stream</nm>
            </application>
         </expression>
      </replacement>
   </constructor>

   <constructor>
      <nm>delta</nm>
      <type><nm>number</nm></type>
      <replacement>
         <expression>
            <nm>_</nm>
            <application>
               <nm>system</nm>
               <nm>command</nm>
            </application>
            <application>
               <nm>read</nm>
               <lit t="string">buffer.$</lit>
            </application>
            <application>
               <nm>-</nm>
               <nm>before</nm>
            </application>
         </expression>
      </replacement>
   </constructor>

   <by>
      <replacement>
         <expression>
            <nm>scope</nm>
            <application>
               <nm>udot</nm>
               <nm>file_name</nm>
            </application>
            <application>
               <nm>udot</nm>
               <lit t="string">: translation took </lit>
            </application>
            <application>
               <nm>dot</nm>
               <nm>delta</nm>
            </application>
            <application>
               <nm>udot</nm>
               <lit t="string"> s.</lit>
            </application>
            <application>
               <nm>newline</nm>
            </application>
         </expression>
      </replacement>
   </by>
</rule>

<!--
   Define a função literal, que traduz os literais
-->

<xsl:comment>Função que traduz os literais (literal)</xsl:comment>
<rule apply="once">
   <nm>show_literal</nm>
   <arg><nm>lit_type</nm><type><nm>stringlit</nm></type></arg>
   <arg><nm>p</nm><type><nm>any</nm></type></arg>

   <xsl:copy-of select="$replace-clause"/>

   <constructor>
```

```
      <nm>str</nm>
      <type><nm>stringlit</nm></type>
      <replacement>
         <expression>
            <nm>_</nm>
            <application><nm>quote</nm><nm>p</nm> </application>
            <application>
               <nm>fix_if_quoted_charlit</nm>
               <nm>lit_type</nm>
            </application>
         </expression>
      </replacement>
   </constructor>

   <constructor>
      <nm>str_list</nm>
      <type mod="*"><nm>stringlit</nm></type>
      <replacement>
         <expression>
            <nm>_</nm>
            <application><nm>.</nm><nm>str</nm></application>
            <application><nm>convert_special_chars</nm></application>
         </expression>
      </replacement>
   </constructor>

   <constructor>
      <nm>result</nm>
      <type mod="*"><nm>token</nm></type>
      <replacement>
         <expression>
            <nm>_</nm>
            <application><nm>udot</nm><each><nm>str_list</nm></each></application>
         </expression>
      </replacement>
   </constructor>

   <constructor>
      <nm>lit_tag</nm>
      <type><nm>stringlit</nm></type>
      <replacement>
         <expression>
            <nm>_</nm>
            <application><nm>+</nm><lit t="string">&lt;</lit></application>
            <application><nm>+</nm><nm>lit_type</nm></application>
            <application><nm>+</nm><lit t="string">&gt;</lit></application>
         </expression>
      </replacement>
   </constructor>

   <by>
      <replacement>
         <expression>
            <nm>output</nm>
            <application><nm>udot</nm><nm>lit_tag</nm></application>
            <application><nm>.</nm><nm>result</nm></application>
            <application><nm>lux_end</nm><nm>lit_type</nm></application>
         </expression>
      </replacement>
   </by>
</rule>


<xsl:comment>Função que corrige literais com aspas </xsl:comment>
<rule apply="once">
   <nm>fix_if_quoted_charlit</nm>
   <arg><nm>lit_type</nm><type><nm>stringlit</nm></type></arg>

   <replace>
      <type><nm>stringlit</nm></type>
      <pattern><nm>s</nm><type><nm>stringlit</nm></type></pattern>
   </replace>

   <condition mod="where">
      <expression>
         <nm>lit_type</nm>
```

```
            <application><nm>=</nm><lit t="string">stringlit</lit></application>
            <application><nm>=</nm><lit t="string">charlit</lit></application>
            <application><nm>=</nm><lit t="string">quote</lit></application>
         </expression>
      </condition>

      <by>
         <replacement>
            <expression>
               <nm>s</nm>
               <application><nm>fix_quoted_charlit</nm></application>
            </expression>
         </replacement>
      </by>
   </rule>

</xsl:template>

<!--=========================================================================================-->

<!--
   Gabarito para derivações simples. A função tradutora é escrita diretamente.
-->
<xsl:template match="define[count(derivation)=1]">
   <rule apply="once">
      <nm><xsl:value-of select="$grammar"/>_<xsl:value-of select="nm"/></nm>
      <arg><nm>p</nm><type><nm><xsl:value-of select="nm"/></nm></type></arg>
      <deconstructor>
         <nm>p</nm>
         <pattern>
            <xsl:for-each select="derivation/*">
               <xsl:choose>
                  <xsl:when test="name()='lit'">
                     <xsl:copy-of select="."/>
                  </xsl:when>
                  <xsl:otherwise>   <!-- name()='type' -->
                     <nm>v<xsl:number/></nm>
                     <xsl:copy-of select="."/>
                  </xsl:otherwise>
               </xsl:choose>
            </xsl:for-each>
         </pattern>
      </deconstructor>
      <xsl:copy-of select="$replace-clause"/>
      <by>
         <replacement>
            <expression>
               <nm>output</nm>
               <application>
                  <nm>udot</nm>
                  <lit t="string">&lt;<xsl:value-of select="nm"/>&gt;</lit>
               </application>
               <application>
                  <nm>newline</nm>
               </application>
               <xsl:apply-templates select="derivation"/>
               <application>
                  <nm>lux_end</nm>
                  <lit t="string"><xsl:value-of select="nm"/></lit>
               </application>
               <application>
                  <nm>smart_flush</nm>
               </application>
            </expression>
         </replacement>
      </by>
   </rule>
</xsl:template>

<!--=========================================================================================-->

<!--
   Gabarito para derivações múltiplas. É necessário gerar funções intermediárias
   mutuamente exclusivas para cada regra e depois aplicá-las em seqüência.
-->
<xsl:template match="define[count(derivation)>1]">
```

```
   <xsl:comment>multiple: <xsl:value-of select="nm"/></xsl:comment>

   <!-- Primeiro, cria uma função que aplica seqüencialmente as regras de tradução de cada
derivação -->

   <xsl:variable name="qname" select="translate(nm, '_', '-')"/>

   <rule apply="once">
      <nm><xsl:value-of select="$grammar"/>_<xsl:value-of select="nm"/></nm>
      <arg><nm>p</nm><type><nm><xsl:value-of select="nm"/></nm></type></arg>
      <xsl:copy-of select="$replace-clause"/>
      <by>
         <replacement>
            <expression>
               <nm>output</nm>
               <application>
                  <nm>udot</nm>
                  <lit t="string">&lt;<xsl:value-of select="nm"/>&gt;</lit>
               </application>
               <application>
                  <nm>newline</nm>
               </application>
               <xsl:for-each select="derivation">
                  <application>
                     <nm>
                        <xsl:value-of select="$grammar"/>
                        <xsl:text>_</xsl:text>
                        <xsl:value-of select="../nm"/>
                        <xsl:text>_</xsl:text>
                        <xsl:number format="A"/>
                     </nm>
                     <nm>p</nm>
                  </application>
               </xsl:for-each>
               <application>
                  <nm>lux_end</nm>
                  <lit t="string"><xsl:value-of select="nm"/></lit>
               </application>
               <application>
                  <nm>smart_flush</nm>
               </application>
            </expression>
         </replacement>
      </by>
   </rule>

   <!-- Depois, cria funções mutuamente exclusivas para cada derivação separadamente. -->

   <xsl:for-each select="derivation">
      <rule apply="once">
         <nm>
            <xsl:value-of select="$grammar"/>
            <xsl:text>_</xsl:text>
            <xsl:value-of select="../nm"/>
            <xsl:text>_</xsl:text>
            <xsl:number format="A"/>
         </nm>
         <arg><nm>p</nm><type><nm><xsl:value-of select="../nm"/></nm></type></arg>

         <deconstructor>
            <nm>p</nm>
            <pattern>
               <xsl:for-each select="*">
                  <xsl:choose>
                     <xsl:when test="name()='lit'">
                        <xsl:copy-of select="."/>
                     </xsl:when>
                     <xsl:otherwise>   <!-- name()='type' -->
                        <nm>v<xsl:number/></nm>
                        <xsl:copy-of select="."/>
                     </xsl:otherwise>
                  </xsl:choose>
               </xsl:for-each>
            </pattern>
         </deconstructor>
```

```
         <xsl:copy-of select="$replace-clause"/>

         <xsl:if test="position() > 1">
            <condition mod="not">
               <expression>
                  <nm>output</nm>
                  <xsl:for-each select="preceding-sibling::derivation">
                     <application mod="?">
                        <nm>
                           <xsl:value-of select="$grammar"/>
                           <xsl:text>_</xsl:text>
                           <xsl:value-of select="../nm"/>
                           <xsl:text>_</xsl:text>
                           <xsl:number format="A"/>
                        </nm>
                        <nm>p</nm>
                     </application>
                  </xsl:for-each>
               </expression>
            </condition>
         </xsl:if>

         <by>
            <replacement>
               <expression>
                  <nm>output</nm><xsl:apply-templates select="."/>
               </expression>
            </replacement>
         </by>

      </rule>
   </xsl:for-each>

</xsl:template>


<!--=========================================================================================-->

<!--
   Este gabarito cria a seqüência de aplicações correspondentes à tradução duma derivação.
-->
<xsl:template match="derivation">
   <xsl:for-each select="*">
      <xsl:choose>
         <!-- cria as chamadas para os terminais (literais) -->
         <xsl:when test="name()='lit'">
            <application>
               <nm>show_literal</nm>
               <lit t="string"><xsl:value-of select="@t"/></lit>
               <lit t="string"><xsl:value-of select="."/></lit>
            </application>
         </xsl:when>

         <!-- cria as chamadas para os não-terminais -->
         <xsl:otherwise> <!-- name()='type' -->
            <xsl:choose>

               <!-- se o tipo não tem atributo mod, basta invocar sua função de tradução --
>
               <xsl:when test="not(@mod)">
                  <application>
                     <nm><xsl:value-of select="$grammar"/>_<xsl:value-of select="nm"/></nm>
                     <nm>v<xsl:number/></nm>
                  </application>
               </xsl:when>

               <!-- se possui o atributo mod, o tipo é composto e a chamada é diferente -->
               <xsl:otherwise>

                  <!-- cria um nome único para o tipo para evitar ambigüidades -->
                  <xsl:variable name="type-name">
                     <xsl:choose>
                        <xsl:when test="nm"> <!-- tipos com nomes usam <nm> -->
                           <xsl:value-of select="nm"/>
                        </xsl:when>
```

```
                        <xsl:otherwise>     <!-- tipos com <lit> têm nomes manufaturados --
>
                           <!-- inclui o tipo do literal para garantir unicidade -->
                           <xsl:value-of select="lit/@t"/>
                           <!-- expande o conteúdo -->
                           <xsl:call-template name="transliterate">
                              <xsl:with-param name="str" select="lit"/>
                           </xsl:call-template>
                        </xsl:otherwise>
                     </xsl:choose>
                  </xsl:variable>

                  <!-- cria o nome do elemento continente baseado no nome do tipo -->
                  <xsl:variable name="tag-name">
                     <xsl:choose>
                        <xsl:when test="@mod='?'">opt-</xsl:when>
                        <xsl:when test="@mod='*'">repeat-</xsl:when>
                        <xsl:when test="@mod='+'">repeat-1-</xsl:when>
                        <xsl:when test="@mod=','">list-</xsl:when>
                        <xsl:when test="@mod=',+'">list-1-</xsl:when>
                     </xsl:choose>
                     <xsl:value-of select="$type-name"/>
                  </xsl:variable>

                  <!-- cria chamada para o marcador inicial do elemento continente -->
                  <application>
                     <nm>udot</nm>
                     <lit t="string">&lt;<xsl:value-of select="$tag-name"/>&gt;</lit>
                  </application>

                  <!-- cria a chamada de tradução apropriada de acordo com o tipo composto
-->
                  <application>
                     <xsl:choose>
                        <xsl:when test="@mod='?'"> <!-- opt -->
                           <nm><xsl:value-of select="$grammar"/>_opt_<xsl:value-of
select="$type-name"/></nm>
                           <nm>v<xsl:number/></nm>
                        </xsl:when>
                        <xsl:otherwise>              <!-- repeat, repeat+, list, list+ -->
                           <nm><xsl:value-of select="$grammar"/>_<xsl:value-of
select="$type-name"/></nm>
                           <each><nm>v<xsl:number/></nm></each>
                        </xsl:otherwise>
                     </xsl:choose>
                  </application>

                  <!-- cria a chamada para o marcador final -->
                  <application>
                     <nm>lux_end</nm>
                     <lit t="string"><xsl:value-of select="$tag-name"/></lit>
                  </application>

               </xsl:otherwise>
            </xsl:choose>
         </xsl:otherwise>
      </xsl:choose>
   </xsl:for-each>
</xsl:template>

<!--============================================================================-->

<!--
   Escreve as funções de tradução para os tipos modificados com o modificador opt (?).
   Primeiro, seleciona os nomes de todos os tipos modificados e os coloca numa string,
   que é então processada como uma lista.
-->
<xsl:template name="write-opt-types">
   <xsl:comment>Funções para tipos nominais com o modificador opt (?)</xsl:comment>

   <xsl:variable name="named-types">
      <xsl:for-each select="//type[@mod = '?']/nm">
         <xsl:sort/>
         <xsl:value-of select="."/><xsl:text> </xsl:text>
      </xsl:for-each>
   </xsl:variable>
```

```
   <xsl:call-template name="write-opt-nm">
      <xsl:with-param name="list" select="$named-types"/>
   </xsl:call-template>

   <xsl:comment>Funções para tipos literais com o modificador opt (?)</xsl:comment>

   <xsl:variable name="contents">
      <xsl:for-each select="//type[@mod = '?']/lit">
         <xsl:sort select="concat(. , @t)"/>
         <xsl:value-of select="."/><xsl:text> </xsl:text>
      </xsl:for-each>
   </xsl:variable>
   <xsl:variable name="attributes">
      <xsl:for-each select="//type[@mod = '?']/lit">
         <xsl:sort select="concat(. , @t)"/>
         <xsl:value-of select="@t"/><xsl:text> </xsl:text>
      </xsl:for-each>
   </xsl:variable>
   <xsl:call-template name="write-opt-lit">
      <xsl:with-param name="cont-list" select="$contents"/>
      <xsl:with-param name="attr-list" select="$attributes"/>
   </xsl:call-template>

</xsl:template>


<!--
   Escreve as funções de tradução para os itens opcionais com nome.
-->
<xsl:template name="write-opt-nm">
   <xsl:param name="list"/>
   <xsl:if test="$list">
      <xsl:variable name="first" select="substring-before($list, ' ')"/>
      <xsl:variable name="rest" select="substring-after($list, ' ')"/>
      <xsl:variable name="next" select="substring-before($rest, ' ')"/>
      <xsl:if test="$first != $next">
         <rule apply="once">
            <nm>
               <xsl:value-of select="$grammar"/>
               <xsl:text>_opt_</xsl:text>
               <xsl:value-of select="$first"/>
            </nm>
            <arg><nm>p</nm><type mod='?'><nm><xsl:value-of
select="$first"/></nm></type></arg>
            <deconstructor>
               <nm>p</nm>
               <pattern><nm>v</nm><type><nm><xsl:value-of
select="$first"/></nm></type></pattern>
            </deconstructor>
            <xsl:copy-of select="$replace-clause"/>
            <by>
               <replacement>
                  <expression>
                     <nm>output</nm>
                     <application>
                        <nm><xsl:value-of select="$grammar"/>_<xsl:value-of
select="$first"/></nm>
                        <nm>v</nm>
                     </application>
                     <application><nm>newline</nm></application>
                  </expression>
               </replacement>
            </by>
         </rule>
      </xsl:if>
      <xsl:call-template name="write-opt-nm">
         <xsl:with-param name="list" select="$rest"/>
      </xsl:call-template>
   </xsl:if>
</xsl:template>

<!--
   Escreve as funções de tradução para os literais opcionais.
-->
<xsl:template name="write-opt-lit">
   <xsl:param name="cont-list"/>
```

```
   <xsl:param name="attr-list"/>
   <xsl:if test="$cont-list and $attr-list">
      <xsl:variable name="cont" select="substring-before($cont-list, ' ')"/>
      <xsl:variable name="attr" select="substring-before($attr-list, ' ')"/>
      <xsl:variable name="cont-rest" select="substring-after($cont-list, ' ')"/>
      <xsl:variable name="attr-rest" select="substring-after($attr-list, ' ')"/>
      <xsl:variable name="cont-next" select="substring-before($cont-rest, ' ')"/>
      <xsl:variable name="attr-next" select="substring-before($attr-rest, ' ')"/>
      <xsl:if test="$cont != $cont-next or $attr != $attr-next">
         <!-- create a valid and unique TXL name for the translation function -->
         <xsl:variable name="translit">
            <xsl:value-of select="concat($grammar, '_opt_', $attr)"/>
            <xsl:call-template name="transliterate">
               <xsl:with-param name="str" select="$cont"/>
            </xsl:call-template>
         </xsl:variable>
         <!-- output the function itself -->
         <rule apply="once">
            <nm><xsl:value-of select="$translit"/></nm>
            <arg>
               <nm>p</nm>
               <type mod='?'><lit t="{$attr}"><xsl:value-of select="$cont"/></lit></type>
            </arg>
            <deconstructor>
               <nm>p</nm>
               <pattern>
                  <lit t="{$attr}"><xsl:value-of select="$cont"/></lit>
               </pattern>
            </deconstructor>
            <xsl:copy-of select="$replace-clause"/>
            <by>
               <replacement>
                  <expression>
                     <nm>output</nm>
                     <application>
                        <nm>show_literal</nm>
                        <lit t="string"><xsl:value-of select="$attr"/></lit>
                        <lit t="string"><xsl:value-of select="$cont"/></lit>
                     </application>
                     <application><nm>newline</nm></application>
                  </expression>
               </replacement>
            </by>
         </rule>
      </xsl:if>
      <xsl:call-template name="write-opt-lit">
         <xsl:with-param name="cont-list" select="$cont-rest"/>
         <xsl:with-param name="attr-list" select="$attr-rest"/>
      </xsl:call-template>
   </xsl:if>
</xsl:template>


<!--
   Cria uma cadeia composta apenas de caracteres válidos num id que corresponde
   ao nome de um terminal, passado no parâmetro str.
-->
<xsl:template name="transliterate">
   <xsl:param name="str"/>
   <xsl:if test="$str">
      <xsl:variable name="char" select="substring($str, 1, 1)"/>
      <xsl:variable name="rest" select="substring($str, 2)"/>
      <xsl:text>_</xsl:text>
      <xsl:choose>
         <xsl:when
test="contains('0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ', $char)">
            <xsl:value-of select="$char"/>
         </xsl:when>
         <xsl:when test="$char = '!'">excl</xsl:when>
         <xsl:when test="$char = '&quot;'">quot</xsl:when>
         <xsl:when test="$char = '#'">numb</xsl:when>
         <xsl:when test="$char = '$'">cifr</xsl:when>
         <xsl:when test="$char = '%'">perc</xsl:when>
         <xsl:when test="$char = '&amp;'">amp</xsl:when>
         <xsl:when test='$char = "&apos;"'>apos</xsl:when>
         <xsl:when test="$char = '('">oparen</xsl:when>
```

```
         <xsl:when test="$char = ')'">cparen</xsl:when>
         <xsl:when test="$char = '*'">star</xsl:when>
         <xsl:when test="$char = '+'">plus</xsl:when>
         <xsl:when test="$char = ','">comma</xsl:when>
         <xsl:when test="$char = '-'">hyphen</xsl:when>
         <xsl:when test="$char = '.'">dot</xsl:when>
         <xsl:when test="$char = '/'">slash</xsl:when>
         <xsl:when test="$char = ':'">colon</xsl:when>
         <xsl:when test="$char = ';'">semicolon</xsl:when>
         <xsl:when test="$char = '&lt;'">lt</xsl:when>
         <xsl:when test="$char = '='">eq</xsl:when>
         <xsl:when test="$char = '&gt;'">gt</xsl:when>
         <xsl:when test="$char = '?'">int</xsl:when>
         <xsl:when test="$char = '@'">at</xsl:when>
         <xsl:when test="$char = '['">obrack</xsl:when>
         <xsl:when test="$char = '\'">bslash</xsl:when>
         <xsl:when test="$char = ']'">cbrack</xsl:when>
         <xsl:when test="$char = '^'">circ</xsl:when>
         <xsl:when test="$char = '_'">unds</xsl:when>
         <xsl:when test="$char = '`'">grave</xsl:when>
         <xsl:when test="$char = '{'">obrace</xsl:when>
         <xsl:when test="$char = '|'">vbar</xsl:when>
         <xsl:when test="$char = '}'">cbrace</xsl:when>
         <xsl:when test="$char = '~'">tilde</xsl:when>
         <xsl:otherwise>
            <xsl:message>Could not transliterate char(<xsl:value-of select="$char"/>).
Invalid function name generated.</xsl:message>
            <xsl:value-of select="$char"/>
         </xsl:otherwise>
      </xsl:choose>
      <xsl:call-template name="transliterate">
         <xsl:with-param name="str" select="$rest"/>
      </xsl:call-template>
   </xsl:if>
</xsl:template>


<!--========================================================================================-->

<!--
   Processa as definições de tokens.
-->
<xsl:template name="write-token-types">

   <xsl:variable name="token-types">
      <xsl:for-each select="//token-def/nm">
         <xsl:sort/>
         <xsl:value-of select="."/><xsl:text> </xsl:text>
      </xsl:for-each>
      <xsl:text>id number charlit stringlit comment upperlowerid upperid loweruppperid
lowerid floatnumber decimalnumber integernumber empty key token </xsl:text>
   </xsl:variable>
   <xsl:call-template name="write-token">
      <xsl:with-param name="list" select="$token-types"/>
   </xsl:call-template>

</xsl:template>


<!--
   Escreve as funções de tradução para os tokens da lista.
-->
<xsl:template name="write-token">
   <xsl:param name="list"/>
   <xsl:if test="$list">
      <xsl:variable name="first" select="substring-before($list, ' ')"/>
      <xsl:variable name="rest" select="substring-after($list, ' ')"/>
      <xsl:if test="not(contains(concat(' ',$rest), concat(' ',$first,' ')))">
         <xsl:comment>token: <xsl:value-of select="$first"/></xsl:comment>
         <rule apply="once">
            <nm><xsl:value-of select="$grammar"/>_<xsl:value-of select="$first"/></nm>
            <arg><nm>p</nm><type><nm><xsl:value-of select="$first"/></nm></type></arg>
            <xsl:copy-of select="$replace-clause"/>
            <by>
               <replacement>
                  <expression>
```

```
                     <nm>output</nm>
                     <application>
                        <nm>show_literal</nm>
                        <lit t="string"><xsl:value-of select="$first"/></lit>
                        <nm>p</nm>
                     </application>
                     <application><nm>newline</nm></application>
                  </expression>
               </replacement>
            </by>
         </rule>
      </xsl:if>
      <xsl:call-template name="write-token">
         <xsl:with-param name="list" select="$rest"/>
      </xsl:call-template>
   </xsl:if>
</xsl:template>

<!--=========================================================================-->

<!--
   Escreve funções de tradução para os tokens pré-definidos (como SP).
-->
<xsl:template name="write-predef-types">

   <xsl:call-template name="write-predef">
      <xsl:with-param name="list" select="'NL EX IN SP SPON SPOFF '"/>
   </xsl:call-template>

</xsl:template>


<!--  Escreve as funções de tradução para os tokens da lista. -->
<xsl:template name="write-predef">
   <xsl:param name="list"/>
   <xsl:if test="$list">
      <xsl:variable name="first" select="substring-before($list, ' ')"/>
      <xsl:variable name="rest" select="substring-after($list, ' ')"/>
      <xsl:comment>predef type: <xsl:value-of select="$first"/></xsl:comment>
      <rule apply="once">
         <nm><xsl:value-of select="$grammar"/>_<xsl:value-of select="$first"/></nm>
         <arg><nm>p</nm><type><nm><xsl:value-of select="$first"/></nm></type></arg>
         <xsl:copy-of select="$replace-clause"/>
         <by>
            <replacement>
               <expression>
                  <nm>output</nm>
                  <application>
                     <nm>udot</nm>
                     <lit t="string">&lt;<xsl:value-of select="$first"/>/&gt;</lit>
                  </application>
                  <application><nm>newline</nm></application>
               </expression>
            </replacement>
         </by>
      </rule>
      <xsl:call-template name="write-predef">
         <xsl:with-param name="list" select="$rest"/>
      </xsl:call-template>
   </xsl:if>
</xsl:template>

<!--=========================================================================-->

<!--
   Gabarito default. Todos os outros elementos que não são defines são ignorados.
-->
<xsl:template match="*">
</xsl:template>

</xsl:transform>
<!-- fim de txl_gentree.xslt -->
```

txl_genproto.xslt

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--
   PUC-Rio - Departamento de Informática

   Arquivo:    txl_genproto.xslt
   Projeto:    Lux
   Conteúdo:   Gerador de protótipos de tradutores
   Autor:      Marcelo Jaccoud Amaral
   Data:       2/5/2001

   Descrição:
      Esta transformação gera o esqueleto de um programa tradutor Lux. Um tradutor é um
      programa TXL que importa código fonte baseado em uma gramática qualquer e gera
      um equivalemte Lux (um arquivo XML com o fonte e um RDF com os comentários).
      O arquivo é gerado no formato Lux. Para obter o arquivo texto correspondente,
      use a transformação txl_tangle, que constrói um arquivo texto a partir do
      fonte TXL-Lux.
      O programa gerado não é plenamente funcional, mas possui a definição de todas as
      regras básicas, permitindo traduzir individualmente cada elemento da árvore
      sintática definido na gramática TXL da linguagem de programação.


-->

<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >
<xsl:param name="grammar" select="'xyz'"/>
   <!--
      $grammar é o nome da gramática para a qual o tradutor será gerado.
      Este nome deve ser o nome (sem a extensão) do arquivo DTD criado para linguagem.
      Esta mesma cadeia será usada como prefixo para o nome de cada função, para indicar
      a linguagem; por  exemplo, ao gerar um programa tradutor para fortran-77, $grammar
      pode ser 'f77'. O programa gerado usará f77.dtd e todas as funções do método
      da gramática achatada terão seus nomes iniciados por f77_ seguidos do nome do
      não-terminal correspondente.
   -->

<xsl:output method="xml" encoding="ISO-8859-1" doctype-system="txl.dtd" indent="yes"/>

<!--=============================================================================-->

<!-- Constantes auxiliares -->

<!-- trecho equivalente à claúsula replace, que se repete muito nas funções geradas -->

<xsl:variable name="replace-clause">
   <replace>
      <type mod="*"><nm>token</nm></type>
      <pattern><nm>output</nm><type mod="*"><nm>token</nm></type></pattern>
   </replace>
</xsl:variable>

<!--=============================================================================-->

<xsl:template match="program">
<!--
   Gabarito principal, processa <program></program>, o elemento-raiz do documento,
   no estilo 'pull'.    Constrói a parte comum a todos os programas tradutores e
   depois cria funções de tradução para cada elemento   definido na gramática:
   não-terminais (defines), não-terminais opcionais (opt) e terminais (tokens),
   nesta ordem.
-->
   <program>
      <!--
         Primeiro, cria a parte comum do programa
      -->
      <xsl:call-template name="write-common-part"/>
      <!--
         Invoca apply-templates, o que irá desencadear a escrita exaustiva de funções
         tradutoras para todos os elementos, na ordem em que os defines aparecem na
         gramática.São processados todos os elementos define e tokens da gramática.
      -->
      <xsl:comment>Funções correspondentes aos não-terminais</xsl:comment>
```

```
      <xsl:apply-templates/>
      <!--
         Invoca write-opt-types, para escrever as funções tradutoras de todos os
         tipos modificados com 'opt'.
      -->
      <xsl:call-template name="write-opt-types"/>
   </program>
</xsl:template>

<!--=============================================================================-->

<!--
   Escreve a parte comum a todos os programas de tradução, como declarações de
   inclusão e as funções principais.
-->
<xsl:template name="write-common-part">
   <!-- Estas declarações de inclusão se repetem em todos os programas de tradução -->
   <xsl:comment>Declarações de inclusão</xsl:comment>
   <include><lit t="string">lib\TxlExtrn</lit></include>
   <include><lit t="string">lib\System</lit></include>
   <include><lit t="string">lux_fg.txl</lit></include>
   <include><lit t="string">lux_strings.txl</lit></include>
   <include><lit t="string">lux_common.txl</lit></include>
   <include><lit t="string"><xsl:value-of select="$grammar"/>.grm</lit></include>

   <!--
      Define a função mainRule, obrigatória em todo programa TXL.
      Esta lê o nome do arquivo de uma variável do sistema e invoca a função que o traduz.
   -->

   <xsl:comment>Função principal (mainRule)</xsl:comment>
   <rule apply="once">
      <nm>mainRule</nm>

      <replace>
         <type><nm>program</nm></type>
         <pattern><nm>p</nm><type><nm>program</nm></type></pattern>
      </replace>

      <constructor>
         <nm>enviroment</nm>
         <type><nm>stringlit</nm></type>
         <replacement>
            <expression>
               <nm>_</nm>
               <application>
                  <nm>pipe</nm>
                  <lit t="string">%lux%\\getenv LUX_FILE</lit>
               </application>
            </expression>
         </replacement>
      </constructor>

      <constructor>
         <nm>file_name</nm>
         <type><nm>stringlit</nm></type>
         <replacement>
            <expression>
               <nm>_</nm>
               <application>
                  <nm>parse</nm>
                  <nm>enviroment</nm>
               </application>
            </expression>
         </replacement>
      </constructor>

      <constructor>
         <nm>log</nm>
         <type mod="*"><nm>token</nm></type>
         <replacement>
            <expression>
               <nm>_</nm>
               <application>
                  <nm>translate</nm>
                  <nm>file_name</nm>
```

```
                <nm>p</nm>
             </application>
             <application>
                <nm>print</nm>
             </application>
          </expression>
       </replacement>
    </constructor>

    <by><replacement/></by>

</rule>

<!--
   Define a função translate, que cria os arquivos e dispara a cadeia de tradução
-->

<xsl:comment>Função que dispara a tradução (translate)</xsl:comment>
<rule apply="once">
   <nm>translate</nm>
   <arg><nm>file_name</nm><type><nm>stringlit</nm></type></arg>
   <arg><nm>p</nm><type><nm>program</nm></type> </arg>

   <replace>
      <type mod="*"><nm>token</nm></type>
      <pattern>
         <nm>scope</nm>
         <type mod="*"><nm>token</nm></type>
      </pattern>
   </replace>

   <constructor>
      <nm>command</nm>
      <type><nm>stringlit</nm></type>
      <replacement>
         <expression>
            <nm>_</nm>
            <application>
               <nm>+</nm>
               <lit t="string">%lux%\\systime &gt;buffer.$</lit>
            </application>
         </expression>
      </replacement>
   </constructor>

   <constructor>
      <nm>before</nm>
      <type><nm>number</nm></type>
      <replacement>
         <expression>
            <nm>_</nm>
            <application>
               <nm>system</nm>
               <nm>command</nm>
            </application>
            <application>
               <nm>read</nm>
               <lit t="string">buffer.$</lit>
            </application>
         </expression>
      </replacement>
   </constructor>

   <constructor>
      <nm>xml_name</nm>
      <type><nm>stringlit</nm></type>
      <replacement>
         <expression>
            <nm>_</nm>
            <application>
               <nm>+</nm>
               <nm>file_name</nm>
            </application>
            <application>
               <nm>+</nm>
               <lit t="string">.xml</lit>
```

```
            </application>
         </expression>
      </replacement>
   </constructor>

   <constructor>
      <nm>rdf_name</nm>
      <type><nm>stringlit</nm></type>
      <replacement>
         <expression>
            <nm>_</nm>
            <application>
               <nm>+</nm>
               <nm>file_name</nm>
            </application>
            <application>
               <nm>+</nm>
               <lit t="string">.rdf</lit>
            </application>
         </expression>
      </replacement>
   </constructor>

   <constructor>
      <nm>workspace</nm>
      <type mod="*"><nm>token</nm></type>
      <replacement>
         <expression>
            <nm>_</nm>
            <application>
               <nm>lux_start_rdf</nm>
               <nm>file_name</nm>
               <lit t="string">x-<xsl:value-of select="$grammar"/></lit>
               <nm>xml_name</nm>
            </application>
            <application>
               <nm>initiate_stream</nm>
               <nm>xml_name</nm>
            </application>
            <application>
               <nm>lux_xmldecl</nm>
            </application>
            <application>
               <nm>lux_stylesheet</nm>
               <lit t="string">text/css</lit>
               <lit t="string"><xsl:value-of select="$grammar"/>.css</lit>
            </application>
            <application>
               <nm>lux_doctype</nm>
               <lit t="string">program</lit>
               <lit t="string"><xsl:value-of select="$grammar"/>.dtd</lit>
            </application>
            <application>
               <nm><xsl:value-of select="$grammar"/>_program</nm>
               <nm>p</nm>
            </application>
            <application>
               <nm>close_stream</nm>
            </application>
            <application>
               <nm>lux_end_rdf</nm>
            </application>
            <application>
               <nm>lux_copy_rdf</nm>
               <nm>rdf_name</nm>
            </application>
         </expression>
      </replacement>
   </constructor>

   <constructor>
      <nm>delta</nm>
      <type><nm>number</nm></type>
      <replacement>
         <expression>
            <nm>_</nm>
```

```
               <application>
                  <nm>system</nm>
                  <nm>command</nm>
               </application>
               <application>
                  <nm>read</nm>
                  <lit t="string">buffer.$</lit>
               </application>
               <application>
                  <nm>-</nm>
                  <nm>before</nm>
               </application>
            </expression>
         </replacement>
      </constructor>

      <by>
         <replacement>
            <expression>
               <nm>scope</nm>
               <application>
                  <nm>udot</nm>
                  <nm>file_name</nm>
               </application>
               <application>
                  <nm>udot</nm>
                  <lit t="string">: translation took </lit>
               </application>
               <application>
                  <nm>dot</nm>
                  <nm>delta</nm>
               </application>
               <application>
                  <nm>udot</nm>
                  <lit t="string"> s.</lit>
               </application>
               <application>
                  <nm>newline</nm>
               </application>
            </expression>
         </replacement>
      </by>
   </rule>
</xsl:template>

<!--=========================================================================================-->

<!--
   Gabarito para derivações simples. A função tradutora é escrita diretamente.
-->
<xsl:template match="define[count(derivation)=1]">
   <xsl:comment>Simple derivation: <xsl:value-of select="nm"/></xsl:comment>

   <xsl:variable name="qname" select="translate(nm, '_', '-')"/>

   <rule apply="once">
      <nm><xsl:value-of select="$grammar"/>_<xsl:value-of select="nm"/></nm>
      <arg><nm>p</nm><type><nm><xsl:value-of select="nm"/></nm></type></arg>
      <deconstructor>
         <nm>p</nm>
         <pattern>
            <xsl:for-each select="derivation/*">
               <xsl:choose>
                  <xsl:when test="name()='lit'">
                     <xsl:copy-of select="."/>
                  </xsl:when>
                  <xsl:otherwise>   <!-- name()='type' -->
                     <nm>v<xsl:number/></nm>
                     <xsl:copy-of select="."/>
                  </xsl:otherwise>
               </xsl:choose>
            </xsl:for-each>
         </pattern>
      </deconstructor>
      <xsl:copy-of select="$replace-clause"/>
      <by>
```

```
         <replacement>
            <expression>
               <nm>output</nm>
               <application>
                  <nm>lux_start</nm>
                  <lit t="string"><xsl:value-of select="$qname"/></lit>
               </application>
               <xsl:apply-templates select="derivation"/>
               <application>
                  <nm>lux_end</nm>
                  <lit t="string"><xsl:value-of select="$qname"/></lit>
               </application>
            </expression>
         </replacement>
      </by>
   </rule>
</xsl:template>

<!--=========================================================================================-->

<!--
   Gabarito para derivações múltiplas. É necessário gerar funções intermediárias
   mutuamente exclusivas para cada regra e depois aplicá-las em seqüência.
-->
<xsl:template match="define[count(derivation)>1]">
   <xsl:comment>multiple: <xsl:value-of select="nm"/></xsl:comment>

   <!-- Primeiro, cria uma função que aplica seqüencialmente as regras de tradução de cada
derivação -->

   <xsl:variable name="qname" select="translate(nm, '_', '-')"/>

   <rule apply="once">
      <nm><xsl:value-of select="$grammar"/>_<xsl:value-of select="nm"/></nm>
      <arg><nm>p</nm><type><nm><xsl:value-of select="nm"/></nm></type></arg>
      <xsl:copy-of select="$replace-clause"/>
      <by>
         <replacement>
            <expression>
               <nm>output</nm>
               <application>
                  <nm>lux_start</nm>
                  <lit t="string"><xsl:value-of select="$qname"/></lit>
               </application>
               <xsl:for-each select="derivation">
                  <application>
                     <nm>
                        <xsl:value-of select="$grammar"/>
                        <xsl:text>_</xsl:text>
                        <xsl:value-of select="../nm"/>
                        <xsl:text>_</xsl:text>
                        <xsl:number format="A"/>
                     </nm>
                     <nm>p</nm>
                  </application>
               </xsl:for-each>
               <application>
                  <nm>lux_end</nm>
                  <lit t="string"><xsl:value-of select="$qname"/></lit>
               </application>
            </expression>
         </replacement>
      </by>
   </rule>

   <!-- Depois, cria funções mutuamente exclusivas para cada derivação separadamente. -->

   <xsl:for-each select="derivation">
      <rule apply="once">
         <nm>
            <xsl:value-of select="$grammar"/>
            <xsl:text>_</xsl:text>
            <xsl:value-of select="../nm"/>
            <xsl:text>_</xsl:text>
            <xsl:number format="A"/>
         </nm>
```

```
         <arg><nm>p</nm><type><nm><xsl:value-of select="../nm"/></nm></type></arg>

         <deconstructor>
            <nm>p</nm>
            <pattern>
               <xsl:for-each select="*">
                  <xsl:choose>
                     <xsl:when test="name()='lit'">
                        <xsl:copy-of select="."/>
                     </xsl:when>
                     <xsl:otherwise>   <!-- name()='type' -->
                        <nm>v<xsl:number/></nm>
                        <xsl:copy-of select="."/>
                     </xsl:otherwise>
                  </xsl:choose>
               </xsl:for-each>
            </pattern>
         </deconstructor>

         <xsl:copy-of select="$replace-clause"/>

         <xsl:if test="position() > 1">
            <condition mod="not">
               <expression>
                  <nm>output</nm>
                  <xsl:for-each select="preceding-sibling::derivation">
                     <application mod="?">
                        <nm>
                           <xsl:value-of select="$grammar"/>
                           <xsl:text>_</xsl:text>
                           <xsl:value-of select="../nm"/>
                           <xsl:text>_</xsl:text>
                           <xsl:number format="A"/>
                        </nm>
                        <nm>p</nm>
                     </application>
                  </xsl:for-each>
               </expression>
            </condition>
         </xsl:if>

         <by>
            <replacement>
               <expression>
                  <nm>output</nm><xsl:apply-templates select="."/>
               </expression>
            </replacement>
         </by>

      </rule>
   </xsl:for-each>

</xsl:template>


<!--=========================================================================================-->

<!--
   Este gabarito cria a seqüência de aplications correspondentes à tradução duma derivação.
-->
<xsl:template match="derivation">
   <xsl:for-each select="*">
      <application>
         <xsl:choose>
            <xsl:when test="name()='lit'">
               <nm><xsl:value-of select="$grammar"/>_lit</nm>
               <xsl:copy-of select="."/>
            </xsl:when>
            <xsl:otherwise>   <!-- name()='type' -->
               <xsl:choose>
                  <!-- tipos simples, sem atributo mod -->
                  <xsl:when test="not(@mod)">
                     <nm><xsl:value-of select="$grammar"/>_<xsl:value-of select="nm"/></nm>
                     <nm>v<xsl:number/></nm>
                  </xsl:when>
```

```
                  <!-- tipos complexos, com modificadores (opt, repeat, etc.) -->
                  <xsl:otherwise>

                     <!-- cria um nome único para o tipo para evitar ambigüidades -->
                     <xsl:variable name="type-name">
                        <xsl:choose>
                           <xsl:when test="nm"> <!-- tipos com nomes usam <nm> -->
                              <xsl:value-of select="nm"/>
                           </xsl:when>
                           <xsl:otherwise>         <!-- tipos com <lit> têm nomes
manufaturados -->
                              <!-- inclui o tipo do literal para garantir unicidade -->
                              <xsl:value-of select="lit/@t"/>
                              <!-- expande o conteúdo -->
                              <xsl:call-template name="transliterate">
                                 <xsl:with-param name="str" select="lit"/>
                              </xsl:call-template>
                           </xsl:otherwise>
                        </xsl:choose>
                     </xsl:variable>

                     <xsl:choose>
                        <xsl:when test="@mod='?'">
                           <nm><xsl:value-of select="$grammar"/>_opt_<xsl:value-of
select="$type-name"/></nm>
                           <nm>v<xsl:number/></nm>
                        </xsl:when>
                        <xsl:otherwise> <!-- * ou + ou , ou ,+ -->
                           <nm><xsl:value-of select="$grammar"/>_<xsl:value-of
select="$type-name"/></nm>
                           <each><nm>v<xsl:number/></nm></each>
                        </xsl:otherwise>
                     </xsl:choose>
                  </xsl:otherwise>

               </xsl:choose>
            </xsl:otherwise>
         </xsl:choose>
      </application>
   </xsl:for-each>
</xsl:template>

<!--============================================================================-->

<!--
   Escreve as funções de tradução para os tipos modificados com o modificador opt (?).
   Primeiro, seleciona os nomes de todos os tipos modificados e os coloca numa string,
   que é então processada como uma lista.
-->
<xsl:template name="write-opt-types">
   <xsl:comment>Funções para tipos nominais com o modificador opt (?)</xsl:comment>

   <xsl:variable name="named-types">
      <xsl:for-each select="//type[@mod = '?']/nm">
         <xsl:sort/>
         <xsl:value-of select="."/><xsl:text> </xsl:text>
      </xsl:for-each>
   </xsl:variable>
   <xsl:call-template name="write-opt-nm">
      <xsl:with-param name="list" select="$named-types"/>
   </xsl:call-template>

   <xsl:comment>Funções para tipos literais com o modificador opt (?)</xsl:comment>

   <xsl:variable name="contents">
      <xsl:for-each select="//type[@mod = '?']/lit">
         <xsl:sort select="concat(. , @t)"/>
         <xsl:value-of select="."/><xsl:text> </xsl:text>
      </xsl:for-each>
   </xsl:variable>
   <xsl:variable name="attributes">
      <xsl:for-each select="//type[@mod = '?']/lit">
         <xsl:sort select="concat(. , @t)"/>
         <xsl:value-of select="@t"/><xsl:text> </xsl:text>
      </xsl:for-each>
   </xsl:variable>
```

```
   <xsl:call-template name="write-opt-lit">
      <xsl:with-param name="cont-list" select="$contents"/>
      <xsl:with-param name="attr-list" select="$attributes"/>
   </xsl:call-template>

</xsl:template>


<xsl:template name="write-opt-nm">
   <xsl:param name="list"/>
   <xsl:if test="$list">
      <xsl:variable name="first" select="substring-before($list, ' ')"/>
      <xsl:variable name="rest" select="substring-after($list, ' ')"/>
      <xsl:variable name="next" select="substring-before($rest, ' ')"/>
      <xsl:if test="$first != $next">
         <rule apply="once">
            <nm>
               <xsl:value-of select="$grammar"/>
               <xsl:text>_opt_</xsl:text>
               <xsl:value-of select="$first"/>
            </nm>
            <arg><nm>p</nm><type mod='?'><nm><xsl:value-of
select="$first"/></nm></type></arg>
            <deconstructor>
               <nm>p</nm>
               <pattern><nm>v</nm><type><nm><xsl:value-of
select="$first"/></nm></type></pattern>
            </deconstructor>
            <xsl:copy-of select="$replace-clause"/>
            <by>
               <replacement>
                  <expression>
                     <nm>output</nm>
                     <application>
                        <nm><xsl:value-of select="$grammar"/>_<xsl:value-of
select="$first"/></nm>
                        <nm>v</nm>
                     </application>
                  </expression>
               </replacement>
            </by>
         </rule>
      </xsl:if>
      <xsl:call-template name="write-opt-nm">
         <xsl:with-param name="list" select="$rest"/>
      </xsl:call-template>
   </xsl:if>
</xsl:template>

<xsl:template name="write-opt-lit">
   <xsl:param name="cont-list"/>
   <xsl:param name="attr-list"/>
   <xsl:if test="$cont-list and $attr-list">
      <xsl:variable name="cont" select="substring-before($cont-list, ' ')"/>
      <xsl:variable name="attr" select="substring-before($attr-list, ' ')"/>
      <xsl:variable name="cont-rest" select="substring-after($cont-list, ' ')"/>
      <xsl:variable name="attr-rest" select="substring-after($attr-list, ' ')"/>
      <xsl:variable name="cont-next" select="substring-before($cont-rest, ' ')"/>
      <xsl:variable name="attr-next" select="substring-before($attr-rest, ' ')"/>
      <xsl:if test="$cont != $cont-next or $attr != $attr-next">
         <!-- create a valid and unique TXL name for the translation function -->
         <xsl:variable name="translit">
            <xsl:value-of select="concat($grammar, '_opt_', $attr)"/>
            <xsl:call-template name="transliterate">
               <xsl:with-param name="str" select="$cont"/>
            </xsl:call-template>
         </xsl:variable>
         <!-- output the function itself -->
         <rule apply="once">
            <nm><xsl:value-of select="$translit"/></nm>
            <arg>
               <nm>p</nm>
               <type mod='?'><lit t="{$attr}"><xsl:value-of select="$cont"/></lit></type>
            </arg>
            <deconstructor>
               <nm>p</nm>
```

```
               <pattern>
                  <lit t="{$attr}"><xsl:value-of select="$cont"/></lit>
               </pattern>
            </deconstructor>
            <xsl:copy-of select="$replace-clause"/>
            <by>
               <replacement>
                  <expression>
                     <nm>output</nm>
                     <application>
                        <nm>dot</nm>
                        <lit t="{$attr}"><xsl:value-of select="$cont"/></lit>
                     </application>
                  </expression>
               </replacement>
            </by>
         </rule>
      </xsl:if>
      <xsl:call-template name="write-opt-lit">
         <xsl:with-param name="cont-list" select="$cont-rest"/>
         <xsl:with-param name="attr-list" select="$attr-rest"/>
      </xsl:call-template>
   </xsl:if>
</xsl:template>


<xsl:template name="transliterate">
   <xsl:param name="str"/>
   <xsl:if test="$str">
      <xsl:variable name="char" select="substring($str, 1, 1)"/>
      <xsl:variable name="rest" select="substring($str, 2)"/>
      <xsl:text>_</xsl:text>
      <xsl:choose>
         <xsl:when
test="contains('0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ', $char)">
            <xsl:value-of select="$char"/>
         </xsl:when>
         <xsl:when test="$char = '!'">excl</xsl:when>
         <xsl:when test="$char = '&quot;'">quot</xsl:when>
         <xsl:when test="$char = '#'">numb</xsl:when>
         <xsl:when test="$char = '$'">cifr</xsl:when>
         <xsl:when test="$char = '%'">perc</xsl:when>
         <xsl:when test="$char = '&amp;'">amp</xsl:when>
         <xsl:when test='$char = "&apos;"'>apos</xsl:when>
         <xsl:when test="$char = '('">oparen</xsl:when>
         <xsl:when test="$char = ')'">cparen</xsl:when>
         <xsl:when test="$char = '*'">star</xsl:when>
         <xsl:when test="$char = '+'">plus</xsl:when>
         <xsl:when test="$char = ','">comma</xsl:when>
         <xsl:when test="$char = '-'">hyphen</xsl:when>
         <xsl:when test="$char = '.'">dot</xsl:when>
         <xsl:when test="$char = '/'">slash</xsl:when>
         <xsl:when test="$char = ':'">colon</xsl:when>
         <xsl:when test="$char = ';'">semicolon</xsl:when>
         <xsl:when test="$char = '&lt;'">lt</xsl:when>
         <xsl:when test="$char = '='">eq</xsl:when>
         <xsl:when test="$char = '&gt;'">gt</xsl:when>
         <xsl:when test="$char = '?'">int</xsl:when>
         <xsl:when test="$char = '@'">at</xsl:when>
         <xsl:when test="$char = '['">obrack</xsl:when>
         <xsl:when test="$char = '\'">bslash</xsl:when>
         <xsl:when test="$char = ']'">cbrack</xsl:when>
         <xsl:when test="$char = '^'">circ</xsl:when>
         <xsl:when test="$char = '_'">unds</xsl:when>
         <xsl:when test="$char = '`'">grave</xsl:when>
         <xsl:when test="$char = '{'">obrace</xsl:when>
         <xsl:when test="$char = '|'">vbar</xsl:when>
         <xsl:when test="$char = '}'">cbrace</xsl:when>
         <xsl:when test="$char = '~'">tilde</xsl:when>
         <xsl:otherwise>
            <xsl:message>Could not transliterate char(<xsl:value-of select="$char"/>).
Invalid function name generated.</xsl:message>
            <xsl:value-of select="$char"/>
         </xsl:otherwise>
      </xsl:choose>
      <xsl:call-template name="transliterate">
```

```
         <xsl:with-param name="str" select="$rest"/>
      </xsl:call-template>
   </xsl:if>
</xsl:template>


<!--=============================================================================-->

<!-- Este gabarito processa as seções 'tokens' -->
<xsl:template match="tokens">
   <xsl:apply-templates/>
</xsl:template>


<!--
   Escreve as funções de tradução para as definições de tokens.
   Não são geradas funções para os tokens padrões do TXL (charlit, number, etc.).
-->
<xsl:template match="token-def">
   <xsl:comment>token: <xsl:value-of select="nm"/></xsl:comment>

   <rule apply="once">
      <nm>
         <xsl:value-of select="$grammar"/>
         <xsl:text>_</xsl:text>
         <xsl:value-of select="nm"/>
      </nm>
      <arg><nm>p</nm><type><nm><xsl:value-of select="nm"/></nm></type></arg>
      <xsl:copy-of select="$replace-clause"/>
      <by>
         <replacement>
            <expression>
               <nm>output</nm>
               <application>
                  <nm>dot</nm>
                  <nm>p</nm>
               </application>
            </expression>
         </replacement>
      </by>
   </rule>
</xsl:template>

<!--=============================================================================-->

<!--
   Gabarito default. Todos os outros elementos, irrelevantes, são ignorados.
-->
<xsl:template match="*">
</xsl:template>

</xsl:transform>
<!-- fim de txl_genproto.xslt -->
```

for.dtd

```
<!ENTITY % spec-statemet  "(implicit | parameter | format | entry | type-decl |
              common | data | dimension | equivalence | external | intrinsic |
              namelist | save | stmt-function)" >

<!ENTITY % action-stmt  "(assignment | backspace | call | close | computed-go-to |
              continue | cycle | end-file | exit | go-to | if | inquire | open |
              print | read | return | rewind | stop | write | do | end-do |
              arithmetic-if | assign | assigned-go-to | pause)" >

<!ENTITY % exec-statement  "(%action-stmt; | case-construct | do-construct |
              if-construct)" >

<!ENTITY % statement  "(%spec-statemet; | %exec-statement;)" >

<!ENTITY % var  "(nm,subscript?,range?)" >

<!ENTITY % actual-arg  "(expr | %var; | alt-return)" >

<!ENTITY % position-spec  "(unit | iostat | err)" >

<!ENTITY % close-spec  "(unit | iostat | err | status)" >

<!ENTITY % inquire-spec  "(unit | file | iostat | err | exist | opened | number |
              named | name | access | sequential | direct | form | formatted |
              unformatted | recl | next-rec | blank | position | action |
              read-spec | write-spec | read-write-spec | delim | pad)" >

<!ENTITY % connect-spec  "(unit | iostat | err | file | status | access | form |
              recl | blank | position | action | delim | pad | carriage-control)" >

<!ENTITY % io-control-spec  "(io-unit | format-spec | nml | rec | iostat | err |
              end | advance | size | eor)" >

<!ENTITY % block  "(case-construct | do-construct | if-construct | assignment |
              backspace | call | close | computed-go-to | continue | cycle |
              end-file | exit | go-to | if | inquire | open | print | read |
              return | rewind | stop | write | do | end-do | arithmetic-if |
              assign | assigned-go-to | pause | format | data | entry)*" >

<!ELEMENT program  (main-program | block-data | function | subroutine)* >

<!ELEMENT main-program  (nm?,(%statement;)*) >

<!ELEMENT nm  (#PCDATA) >

<!ELEMENT implicit  (implicit-spec*) >

<!ELEMENT implicit-spec  (type,letter-spec+) >

<!ELEMENT type  (integer | real | double-precision | complex | character | logical) >

<!ELEMENT integer   EMPTY  >

<!ELEMENT real   EMPTY  >

<!ELEMENT double-precision   EMPTY  >

<!ELEMENT complex   EMPTY  >

<!ELEMENT character  (length?) >

<!ELEMENT length  (lit | star | expr) >

<!ELEMENT lit  (#PCDATA) >

<!ELEMENT star   EMPTY  >

<!ELEMENT expr  (op?,(lit | %var; | expr),(op,(lit | %var; | expr))*) >

<!ELEMENT op  (#PCDATA) >

<!ELEMENT subscript  (%actual-arg;)* >
```

```
<!ELEMENT alt-return  (lit) >

<!ELEMENT range  (lower?,upper?) >

<!ELEMENT lower  (expr) >

<!ELEMENT upper  (expr) >

<!ELEMENT logical   EMPTY  >

<!ELEMENT letter-spec  (lit,lit?) >

<!ELEMENT parameter  (nm,expr)+ >

<!ELEMENT format  (lit) >

<!ELEMENT entry  (nm,args?) >

<!ELEMENT args  (arg*) >

<!ELEMENT arg  (nm | star) >

<!ELEMENT type-decl  (type,entity+) >

<!ELEMENT entity  (nm,shape*,length?) >

<!--formerly array-spec
-->
<!ELEMENT shape  (star | (expr,(star | expr)?)) >

<!ELEMENT common  (common-group+) >

<!ELEMENT common-group  (nm?,entity+) >

<!ELEMENT data  (data-set+) >

<!ELEMENT data-set  (data-obj+,data-val+) >

<!ELEMENT data-obj  (%var; | data-do) >

<!ELEMENT data-do  ((data-do-obj | data-do)+,nm,expr,expr,expr?) >

<!ELEMENT data-do-obj  (nm,subscript?) >

<!ELEMENT data-val  (repeat?,(nm | lit)) >

<!ELEMENT repeat  (nm | lit) >

<!ELEMENT dimension  (array-decl+) >

<!ELEMENT array-decl  (nm,shape) >

<!ELEMENT equivalence  (equiv-group+) >

<!ELEMENT equiv-group  (%var;,(%var;)+) >

<!ELEMENT external  (nm+) >

<!ELEMENT intrinsic  (nm+) >

<!ELEMENT namelist  (name-group+) >

<!ELEMENT name-group  (nm,nm+) >

<!ELEMENT save  (nm | block-nm)+ >

<!ELEMENT block-nm  (#PCDATA) >

<!ELEMENT stmt-function  (nm+,expr) >

<!ELEMENT assignment  (%var;,expr) >

<!ELEMENT backspace  (expr | (%position-spec;)+) >

<!ELEMENT unit  (expr) >
```

```
<!ELEMENT iostat  %var; >
<!ELEMENT err  (lit) >
<!ELEMENT call  (nm,(%actual-arg;)*) >
<!ELEMENT close  (%close-spec;)+ >
<!ELEMENT status  (expr) >
<!ELEMENT computed-go-to  (lit+,expr) >
<!ELEMENT continue   EMPTY  >
<!ELEMENT cycle   EMPTY  >
<!ELEMENT end-file  (expr | (%position-spec;)+) >
<!ELEMENT exit   EMPTY  >
<!ELEMENT go-to  (lit) >
<!ELEMENT if  (expr,%action-stmt;) >
<!ELEMENT inquire  (%inquire-spec;)+ >
<!ELEMENT file  (expr) >
<!ELEMENT exist  %var; >
<!ELEMENT opened  %var; >
<!ELEMENT number  %var; >
<!ELEMENT named  %var; >
<!ELEMENT name  (expr) >
<!ELEMENT access  (expr) >
<!ELEMENT sequential  (expr) >
<!ELEMENT direct  (expr) >
<!ELEMENT form  (expr) >
<!ELEMENT formatted  (expr) >
<!ELEMENT unformatted  (expr) >
<!ELEMENT recl  (expr) >
<!ELEMENT next-rec  (expr) >
<!ELEMENT blank  (expr) >
<!ELEMENT position  (expr) >
<!ELEMENT action  (expr) >
<!ELEMENT read-spec  (expr) >
<!ELEMENT write-spec  (expr) >
<!ELEMENT read-write-spec  (expr) >
<!ELEMENT delim  (expr) >
<!ELEMENT pad  (expr) >
<!ELEMENT open  (%connect-spec;)+ >
<!ELEMENT carriage-control  (expr) >
<!ELEMENT print  (format-spec,output*) >
```

```
<!ELEMENT format-spec  (lit | star | %var; | expr) >

<!ELEMENT output  (expr | io-do) >

<!ELEMENT io-do  ((input | output)+,nm,expr,expr,expr?) >

<!ELEMENT input  (%var; | io-do) >

<!ELEMENT read  ((format-spec | control-spec),input*) >

<!ELEMENT control-spec  (%io-control-spec;)+ >

<!ELEMENT io-unit  (%var; | star | expr) >

<!ELEMENT nml  (nm) >

<!ELEMENT rec  (expr) >

<!ELEMENT end  (lit) >

<!ELEMENT advance  (expr) >

<!ELEMENT size  (expr) >

<!ELEMENT eor  (lit) >

<!ELEMENT return  (expr?) >

<!ELEMENT rewind  (expr | (%position-spec;)+) >

<!ELEMENT stop  (nm | lit)? >

<!ELEMENT write  (control-spec,output*) >

<!ELEMENT do  (lit?,((%var;,expr,expr,expr?) | expr)?) >

<!ELEMENT end-do   EMPTY  >

<!ELEMENT arithmetic-if  (expr,lit,lit,lit) >

<!ELEMENT assign  (lit,%var;) >

<!ELEMENT assigned-go-to  (%var;,lit*) >

<!ELEMENT pause  (nm | lit)? >

<!ELEMENT case-construct  (expr,case-part*) >

<!ELEMENT case-part  (selector*,%block;) >

<!ELEMENT selector  (expr | (lower,upper?) | upper | default) >

<!ELEMENT default   EMPTY  >

<!ELEMENT do-construct  (do,%block;) >

<!ELEMENT if-construct  (expr,%block;,else-if*,else?,end-if) >

<!ELEMENT else-if  (expr,%block;) >

<!ELEMENT else  %block; >

<!ELEMENT end-if   EMPTY  >

<!ELEMENT block-data  (nm?,(%spec-statemet;)*) >

<!ELEMENT function  (type?,nm,args,(%statement;)*) >

<!ELEMENT subroutine  (nm?,args?,(%statement;)*) >

<!ATTLIST access
              id  ID    #IMPLIED  >

<!ATTLIST action
              id  ID    #IMPLIED  >
```

```
<!ATTLIST advance
              id  ID    #IMPLIED  >

<!ATTLIST alt-return
              id  ID    #IMPLIED  >

<!ATTLIST arg
              id  ID    #IMPLIED  >

<!ATTLIST arithmetic-if
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST array-decl
              id  ID    #IMPLIED  >

<!ATTLIST assign
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST assigned-go-to
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST assignment
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST backspace
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST blank
              id  ID    #IMPLIED  >

<!ATTLIST block-data
              id  ID    #IMPLIED  >

<!ATTLIST call
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST carriage-control
              id  ID    #IMPLIED  >

<!ATTLIST case-construct
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST close
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST common
              id  ID    #IMPLIED  >

<!ATTLIST common-group
              id  ID    #IMPLIED  >

<!ATTLIST complex
              size  CDATA    #IMPLIED  >

<!ATTLIST computed-go-to
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST continue
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST cycle
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST data
```

```
              id  ID    #IMPLIED  >

<!ATTLIST data-do
              id  ID    #IMPLIED  >

<!ATTLIST data-do-obj
              id  ID    #IMPLIED  >

<!ATTLIST data-obj
              id  ID    #IMPLIED  >

<!ATTLIST data-set
              id  ID    #IMPLIED  >

<!ATTLIST data-val
              id  ID    #IMPLIED  >

<!ATTLIST delim
              id  ID    #IMPLIED  >

<!ATTLIST dimension
              id  ID    #IMPLIED  >

<!ATTLIST direct
              id  ID    #IMPLIED  >

<!ATTLIST do
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST do-construct
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST else
              id  ID    #IMPLIED  >

<!ATTLIST else-if
              id  ID    #IMPLIED  >

<!ATTLIST end
              id  ID    #IMPLIED  >

<!ATTLIST end-do
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST end-file
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST end-if
              id  ID    #IMPLIED
              label  CDATA    #IMPLIED  >

<!ATTLIST entity
              id  ID    #IMPLIED  >

<!ATTLIST entry
              id  ID    #IMPLIED  >

<!ATTLIST eor
              id  ID    #IMPLIED  >

<!ATTLIST equiv-group
              id  ID    #IMPLIED  >

<!ATTLIST equivalence
              id  ID    #IMPLIED  >

<!ATTLIST err
              id  ID    #IMPLIED  >

<!ATTLIST exist
              id  ID    #IMPLIED  >
```

```
<!ATTLIST exit
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST expr
              id  ID    #IMPLIED  >

<!ATTLIST external
              id  ID    #IMPLIED  >

<!ATTLIST file
              id  ID    #IMPLIED  >

<!ATTLIST form
              id  ID    #IMPLIED  >

<!ATTLIST format
              label  CDATA    #IMPLIED  >

<!ATTLIST format-spec
              id  ID    #IMPLIED  >

<!ATTLIST formatted
              id  ID    #IMPLIED  >

<!ATTLIST function
              id  ID    #IMPLIED  >

<!ATTLIST go-to
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST if
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST if-construct
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST implicit
              id  ID    #IMPLIED  >

<!ATTLIST input
              id  ID    #IMPLIED  >

<!ATTLIST inquire
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST integer
              size  CDATA    #IMPLIED  >

<!ATTLIST intrinsic
              id  ID    #IMPLIED  >

<!ATTLIST io-do
              id  ID    #IMPLIED  >

<!ATTLIST io-unit
              id  ID    #IMPLIED  >

<!ATTLIST iostat
              id  ID    #IMPLIED  >

<!ATTLIST lit
              t (logical | real | complex | char | boz | letter | free | label |
              integer)   #IMPLIED  >

<!ATTLIST logical
              size  CDATA    #IMPLIED  >

<!ATTLIST main-program
              id  ID    #IMPLIED  >

<!ATTLIST name
```

```
              id  ID    #IMPLIED  >

<!ATTLIST name-group
              id  ID    #IMPLIED  >

<!ATTLIST named
              id  ID    #IMPLIED  >

<!ATTLIST namelist
              id  ID    #IMPLIED  >

<!ATTLIST next-rec
              id  ID    #IMPLIED  >

<!ATTLIST nml
              id  ID    #IMPLIED  >

<!ATTLIST number
              id  ID    #IMPLIED  >

<!ATTLIST open
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST opened
              id  ID    #IMPLIED  >

<!ATTLIST output
              id  ID    #IMPLIED  >

<!ATTLIST pad
              id  ID    #IMPLIED  >

<!ATTLIST parameter
              id  ID    #IMPLIED  >

<!ATTLIST pause
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST position
              id  ID    #IMPLIED  >

<!ATTLIST print
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST program
              id  ID    #IMPLIED  >

<!ATTLIST read
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST read-spec
              id  ID    #IMPLIED  >

<!ATTLIST read-write-spec
              id  ID    #IMPLIED  >

<!ATTLIST real
              size  CDATA    #IMPLIED  >

<!ATTLIST rec
              id  ID    #IMPLIED  >

<!ATTLIST recl
              id  ID    #IMPLIED  >

<!ATTLIST return
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST rewind
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >
```

```
<!ATTLIST save
              id  ID    #IMPLIED  >

<!ATTLIST sequential
              id  ID    #IMPLIED  >

<!ATTLIST shape
              id  ID    #IMPLIED  >

<!ATTLIST size
              id  ID    #IMPLIED  >

<!ATTLIST status
              id  ID    #IMPLIED  >

<!ATTLIST stmt-function
              id  ID    #IMPLIED  >

<!ATTLIST stop
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST subroutine
              id  ID    #IMPLIED  >

<!ATTLIST type
              id  ID    #IMPLIED  >

<!ATTLIST type-decl
              id  ID    #IMPLIED  >

<!ATTLIST unformatted
              id  ID    #IMPLIED  >

<!ATTLIST unit
              id  ID    #IMPLIED  >

<!ATTLIST write
              label  CDATA    #IMPLIED
              id  ID    #IMPLIED  >

<!ATTLIST write-spec
              id  ID    #IMPLIED  >
```

for.grm

```
% PUC-Rio - Departamento de Informática
%=============================================================================
%
% Nome do arquivo:  for.grm
% Projeto:          For-See
% Conteúdo:         Gramática fortran-77 (ISO 1539-1980) modificada para uso
%                    em TXL e estendida para conter algumas construções
%                    adicionais.
% Data:    11/07/2000 08:31
% Autor:   Marcelo Jaccoud Amaral
%
%=============================================================================
%
% Descrição:
%
%  Esta gramática aceita um fortran-77 modificado. O TXL é incapaz de
%  processar diretamente código fortran em virtude do modo peculiar como os
%  espaços em branco e a quebra-de-linha são tratados. Por exemplo, o fortran
%  ignora sumariamente todos os espaços que não pertençam a uma constante
%  literal, ao passo que TXL o toma como delimitador. O fortran entende a
%  quebra-de-linha como um delimitador de comandos, ao passo que TXL a ignora.
%  Para viabilizar o uso de TXL, um pré-processador modifica o código-fonte
%  original e gera um arquivo equivalente numa linguagem modificada. No
%  pré-processamento, as instruções INCLUDE são resolvidas.
%  Esta gramática aceita a linguagem resultante do pré-processamento, que é
%  funcionalmente equivalente à original; somente alguns aspectos léxicos
%  foram modificados.
%  São suportadas todas as construções do fortran definidas no padrão
%  fortran-77 (ISO 1539-1980) e a as seguintes construções adicionais,
%  seguindo o estilo do padrão fortran-90 (ISO 1539-1992):
%     do - end do
%     do while - end do
%     select case - case - end select
%     implicit none
%     cycle
%     exit
%     namelist
%
% Bibliografia:
%  As regras de sintaxe aqui usadas foram extraídas do draft do padrão
%  fortran-90 de Junho de 1990. (O padrão definitivo é o ISO 1539-1992).
%  Naturalmente, as regras não suportadas no fortran-77 foram removidas, mas
%  os nomes dos símbolos não terminais foram mantidos para facilitar uma
%  migração futura.
%=============================================================================

%=================== definições específicas para o TXL =====================

%# pragma -comment
%
% o pragma acima faz com que os comentários sejam significantes, para que
% seu conteúdo possa ser manipulado
% ! é o comentário fortran-90 gerado pelo pré-processador para todos os
%   comentários com C e * do fortran-77
% {} é um comentário especial que estava no interior de uma linha de
%    continuação e portanto aparece como um bloco quando elas foram desfeitas
%    no pré-processamento.

comments
   !
   { }
end comments

compounds
   **  //
   .not. .and. .eqv. .neqv. .true. .false.
   .lt. .le. .eq. .ne. .gt. .ge.
end compounds

% Como fortran não tem palavras reservadas, o comando keys não é usado.

%=============================================================================
```

```
% As seções seguintes seguem a nomenclatura do padrão fortran-90.
% As regras estão identificadas pela sua numeração original.
% As referentes a estruturas não suportadas pelo fortran-77 foram suprimidas,
% bem como as não referenciadas ou incompatíveis com a sintaxe do TXL.
%=============================================================================

% D.1.1 Scope of Standard. ------------------------------------------------

% D.1.2 Fortran Terms and Concepts. ---------------------------------------

define executable_program              % R201
   [program_unit*]
   [comment*]
end define

define program_unit                    % R202
   [main_program]
   | [external_subprogram]
   | [block_data]
end define

define external_subprogram             % R203
   [function_subprogram]
   | [subroutine_subprogram]
end define

define specification_part              % R204
   [implicit_part?] [declaration_construct_block*]
end define

define implicit_part                   % R205
   [implicit_part_block]
   [implicit_stmt]
end define

define implicit_part_block
   [implicit_part_stmt*]
   | [comment*]
end define

define implicit_part_stmt              % R206
   [implicit_stmt]
   | [parameter_stmt]
   | [format_stmt]
   | [entry_stmt]
end define

define declaration_construct_block
   [declaration_construct*]
   | [comment*]
end define

define declaration_construct           % R207
   [type_declaration_stmt]
   | [specification_stmt]
   | [parameter_stmt]
   | [format_stmt]
   | [entry_stmt]
   | [stmt_function_stmt]
end define

define execution_part                  % R208
   [exec_construct] [execution_part_construct_block*]
end define

define execution_part_construct_block
   [execution_part_construct*]
   | [comment*]
end define

define execution_part_construct        % R209
   [exec_construct]
   | [format_stmt]
   | [data_stmt]
   | [entry_stmt]
end define
```

```
define specification_stmt              % R214
   [common_stmt]
   | [data_stmt]
   | [dimension_stmt]
   | [equivalence_stmt]
   | [external_stmt]
   | [intrinsic_stmt]
   | [namelist_stmt]
   | [save_stmt]
end define

define exec_construct
   [label?] [executable_construct]
end define

define executable_construct            % R215
   [action_stmt]                       % do_construct não consta aqui pois
   | [case_construct]                  % não é possível parear os labels
   | [if_construct]                    % v. action_stmt e a seção que trata
end define                             % das instruções DO

define action_stmt                     % R216
   [assignment_stmt]
   | [backspace_stmt]
   | [call_stmt]
   | [close_stmt]
   | [computed_goto_stmt]
   | [continue_stmt]
   | [cycle_stmt]
   | [endfile_stmt]
   | [exit_stmt]
   | [goto_stmt]
   | [if_stmt]
   | [inquire_stmt]
   | [open_stmt]
   | [print_stmt]
   | [read_stmt]
   | [return_stmt]
   | [rewind_stmt]
   | [stop_stmt]
   | [write_stmt]
   % estas instruções são produto do desmonte de do_construct
   | [do_stmt]
   | [end_do_stmt]
   % these four are Fortran-90 obsolescent features
   | [arithmetic_if_stmt]
   | [assign_stmt]
   | [assigned_goto_stmt]
   | [pause_stmt]
end define


% D.1.3 Characteres, Lexical Tokens, and Source Form. -----------------------

tokens
   id          "\a\i*"     % R304 underscore not allowed as first character
   binnumber   "[Bb]'[01]+'"                % R408
   octnumber   "[Oo]'[01234567]+'"          % R409
   hexnumber   "[Zz]'[\dABCDEFabcdef]+'"    % R410
end tokens

define letter
   'a | 'b | 'c | 'd | 'e | 'f | 'g | 'h | 'i | 'j | 'k | 'l | 'm |
   'n | 'o | 'p | 'q | 'r | 's | 't | 'u | 'v | 'w | 'x | 'y | 'z
end define

define comma
   ',
end define

% [id] is used instead of [name] to ease transformations

define constant                        % R305
   [literal_constant]
   | [named_constant]
```

```
end define

define literal_constant                % R306
   [int_literal_constant]
   | [real_literal_constant]
   | [complex_literal_constant]
   | [logical_literal_constant]
   | [char_literal_constant]
   | [boz_literal_constant]
end define

define named_constant                  % R307
   [id]
end define

define int_constant                    % R308
   [constant]                          % must be of type integer
end define

define char_constant                   % R309
   [constant]                          % must be of type character
end define

define label                           % R313
   [integernumber]     % up to 5 digits, at least one must be nonzero
end define


% D.1.4 Intrinsic and Derived Data Types. ----------------------------------

tokens
   floatnumber "[([(\d+.(\d+)?)(.\d+)]([ed][+-]?\d+)?)(\d+[ed][+-]?\d+)]"
   sign        "[+-]"
end tokens

define signed_int_literal_constant     % R403
   [sign?][int_literal_constant]
end define

define int_literal_constant            % R404
   [integernumber]
end define

define boz_literal_constant            % R407
   [binnumber]
   | [octnumber]
   | [hexnumber]
end define

define signed_real_literal_constant    % R412
   [sign?][real_literal_constant]
end define

define real_literal_constant           % R413
   [floatnumber]
end define

define complex_literal_constant        % R417
   '( [real_part] , [imag_part] ')
end define

define real_part                       % R418
   [signed_int_literal_constant]
   | [signed_real_literal_constant]
end define

define imag_part                       % R419
   [signed_int_literal_constant]
   | [signed_real_literal_constant]
end define

define char_literal_constant           % R420
   [charlit+]                          % not Fortran, but TXL stile literals!
end define

define logical_literal_constant        % R421
```

```
   '.true.
   | '.false.
end define


% D.1.5 Data Object Declarations and Specifications. ------------------------

define type_declaration_stmt           % R501
   [type_spec] [entity_decl,+] '; [NL]
end define

define type_spec                       % R502
   'integer [type_size_spec?]
   | 'real [type_size_spec?]
   | 'double 'precision
   | 'complex [type_size_spec?]
   | 'character [char_selector?]
   | 'logical [type_size_spec?]
end define

define type_size_spec                  % not ISO !!!
   '* [int_literal_constant]
end define

define entity_decl                     % R504
   [comment*]
   [object_name] [delim_array_spec?] [char_length_spec?]
   [comment*]
end define

define object_name
   [id]
end define

define delim_array_spec
   '( [array_spec] ')
end define

define char_length_spec
   '* [char_length]
end define

define char_selector
   '( [type_param_value] ')
   | '* [char_length] [comma?]
end define

define char_length                     % R508
   '( [type_param_value] ')
   | [int_literal_constant]
end define

define type_param_value                % R509
   '*
   | [specification_expr]
end define

define array_spec                      % R512
   [last_size_spec]
   | [explicit_shape_spec,+] [last_spec_item?]
end define

define last_spec_item
   ', [last_size_spec]
end define

define explicit_shape_spec             % R513
   [lower_bound] ': [upper_bound]
   | [upper_bound]
end define

define lower_bound                     % R514
   [specification_expr]
end define

define upper_bound                     % R515
```

```
   [specification_expr]
end define

define last_size_spec
   '*
   | [lower_bound] ': '*
end define

define save_stmt                        % R523
   'save [saved_entity,] [comment*] '; [NL]
end define

define saved_entity                     % R524
   [object_name]
   | '/ [common_block_name] '/
end define

define common_block_name
   [id]
end define

define dimension_stmt                   % R525
   'dimension [array_decl,+] '; [comment*] [NL]
end define

define array_decl
   [comment*]
   [array_name] [delim_array_spec]
   [comment*]
end define

define array_name
   [id]
end define

define data_stmt                        % R529
   'data [data_stmt_set] [data_stmt_extra_set*] [comment*] '; [NL]
end define

define data_stmt_set                    % R530
   [data_stmt_object,+] '/ [data_stmt_value,+] '/
end define

define data_stmt_extra_set
   [comma?] [data_stmt_set]
end define

define data_stmt_object                 % R531
   [variable]
   | [data_implied_do]
end define

define data_stmt_value                  % R532
   [data_stmt_repeat_part?] [data_stmt_constant]
end define

define data_stmt_repeat_part
   [data_stmt_repeat] '*
end define

define data_stmt_constant               % R533
   [constant]
   | [signed_int_literal_constant]
   | [signed_real_literal_constant]
   | [boz_literal_constant]
end define

define data_stmt_repeat                 % R534
   [int_constant]
end define

define data_implied_do                  % R535
   '( [data_i_do_object,+] [comma] [data_i_do_variable] '=
   [int_expr] [comma] [int_expr] [data_increment_part?] ')
end define
```

```
define data_increment_part
   [comma] [int_expr]
end define

define data_i_do_object                % R536
   [variable_name] [delim_subscript_list?]
   | [data_implied_do]
end define

define data_i_do_variable              % R537
   [scalar_variable]                   % must be int
end define

define parameter_stmt                  % R538
   'parameter '( [named_constant_def,+] ') [comment*] '; [NL]
end define

define named_constant_def              % R539
   [named_constant] = [initialization_expr]
end define

define implicit_stmt                   % R540
   'implicit [implicit_spec,+] [comment*] '; [NL]
   | 'implicit 'none [comment*] '; [NL]
end define

define implicit_spec                   % R541
   [type_spec] '( [letter_spec,+] ')
end define

define letter_spec                     % R542
   [letter] [letter_spec_cont?]
end define

define letter_spec_cont
   '- [letter]
end define

define namelist_stmt                   % R543
   'namelist [namelist_group] [namelist_extra_group*] [comment*] '; [NL]
end define

define namelist_group
   '/ [namelist_group_name] '/ [namelist_group_object,+]
end define

define namelist_extra_group
   [comma?] [namelist_group]
end define

define namelist_group_name
   [id]
end define

define namelist_group_object           % R544
   [variable_name]
end define

define variable_name
   [id]
end define

define equivalence_stmt                % R545
   'equivalence [equivalence_set,+] [comment*] '; [NL]
end define

define equivalence_set                 % R546
   '( [equivalence_object] ', [equivalence_object,+] ')
end define

define equivalence_object              % R547
   [variable]
end define

define common_stmt                     % R548
   'common [common_block_group] [common_block_extra_group*]
```

```
   [comment*] '; [NL]
end define

define common_block_group
   [common_block_name_part?] [common_block_object,+]
end define

define common_block_extra_group
   [comma?] [common_block_group]
end define

define common_block_name_part
   '/ [common_block_name] '/
end define

define common_block_object             % R549
   [variable_name] [explicit_shape_spec_part?]
end define

define explicit_shape_spec_part
   '( [explicit_shape_spec,+] ')
end define


% D.1.6 Use of Data Objects. ------------------------------------------------

define scalar_variable
   [variable_name]
end define

define variable                        % R601
   [variable_name] [delim_subscript_list?] [delim_substring_range?]
end define

define delim_subscript_list
  '( [subscript,+] ')
end define

define logical_variable                % R603
   [variable]
end define

define char_variable                   % R605
   [variable]
end define

define int_variable                    % R607
   [variable]
end define

define delim_substring_range
   '( [substring_range] ')
end define

define substring_range                 % R611
   [int_expr?] ': [int_expr?]
end define

define subscript                       % R617
   [int_expr]
end define


% D.1.7 Expressions and Assignment. ------------------------------------------

define primary                         % R701
   [literal_constant]
      % não se pode usar [constant] aqui porque gera ambigüidade entre
      % [named_constant] e [variable]
   | [variable]
   | [function_reference]
   | '( [expr] ')
end define

define level_1_expr                    % R703
   [primary]
```

```
end define

define mult_operand                    % R705
   [level_1_expr] [mult_part?]
end define

define mult_part
   [power_op] [mult_operand]
end define

define add_operand                      % R706
   [mult_operand] [mult_parcel]
end define

define mult_parcel
   [mult_op] [mult_operand] [mult_parcel]
   | [empty]
end define

define level_2_expr                     % R707 (recusão à esquerda eliminada)
   [add_operand] [add_parcel]
   | [add_op] [add_operand] [add_parcel]
end define

define add_parcel
   [add_op] [add_operand] [add_parcel]
   | [empty]
end define

define power_op                         % R708
   '**
end define

define mult_op                          % R709
   '*
   | '/
end define

define add_op                           % R710
   '+
   | '-
end define

define level_3_expr                     % R711 (recursão à esquerda eliminada)
   [level_2_expr] [concat_parcel]
end define

define concat_parcel
   [concat_op] [level_2_expr] [concat_parcel]
   | [empty]
end define

define concat_op                        % R712
   '//
end define

define level_4_expr                     % R713 (recursão à esquerda eliminada)
   [level_3_expr] [rel_parcel]
end define

define rel_parcel
   [rel_op] [level_3_expr] [rel_parcel]
   | [empty]
end define

define rel_op                           % R714
   '.eq.
   | '.ne.
   | '.lt.
   | '.le.
   | '.gt.
   | '.ge.
end define

define and_operand                      % R715
   [not_op] [level_4_expr]
```

```
   | [level_4_expr]
end define

define or_operand                       % R716 (recursão à esquerda eliminada)
   [and_operand] [and_parcel]
end define

define and_parcel
   [and_op] [and_operand] [and_parcel]
   | [empty]
end define

define equiv_operand                    % R717 (recursão à esquerda eliminada)
   [or_operand] [or_parcel]
end define

define or_parcel
   [or_op] [or_operand] [or_parcel]
   | [empty]
end define

define level_5_expr                     % R718 (recursão à esquerda eliminada)
   [equiv_operand] [equiv_parcel]
end define

define equiv_parcel
   [equiv_op] [equiv_operand] [equiv_parcel]
   | [empty]
end define

define not_op                           % R719
   '.not.
end define

define and_op                           % R720
   '.and.
end define

define or_op                            % R721
   '.or.
end define

define equiv_op                         % R722
   '.eqv.
   | '.neqv.
end define

define expr                             % R723
   [level_5_expr]
end define

define logical_expr                     % R725
   [expr]
end define

define char_expr                        % R726
   [expr]
end define

define int_expr                         % R728
   [expr]
end define

define numeric_expr                     % R729
   [expr]
end define

define initialization_expr              % R730
   [expr]
end define

define specification_expr               % R734
   [int_expr]
end define

define assignment_stmt                  % R735
```

```
   [variable] '= [expr] [comment*] '; [NL]
end define


% D.1.8 Execution Control. ---------------------------------------------------

define block                            % R801
   [execution_part_construct_block*]
end define

define if_construct                     % R802
   [if_then_stmt]
   [block]
   [else_if_part*]
   [else_part?]
   [end_if_stmt]
end define

define else_if_part
   [else_if_stmt] [block]
end define

define else_part
   [else_stmt] [block]
end define

define if_then_stmt                     % R803
   'if '( [logical_expr] ') 'then [comment*] '; [NL]
end define

define else_if_stmt                     % R804
   'else 'if '( [logical_expr] ') 'then [comment*] '; [NL]
end define

define else_stmt                        % R805
   'else [comment*] '; [NL]
end define

define end_if_stmt                      % R806
   [label?] 'end 'if [comment*] '; [NL]
end define

define if_stmt                          % R807
   'if '( [logical_expr] ') [comment*] [action_stmt]
end define

define case_construct                   % R808
   [select_case_stmt] [comment*] [case_part*] [end_select_stmt]
end define

define case_part
   [case_stmt] [block]
end define

define select_case_stmt                 % R809
   'select 'case '( [case_expr] ') [comment*] '; [NL]
end define

define case_stmt                        % R810
   'case [case_selector] [comment*] '; [NL]
end define

define end_select_stmt                  % R811
   'end 'select [comment*] '; [NL]
end define

define case_expr                        % R812
   [int_expr]
   | [char_expr]
   | [logical_expr]
end define

define case_selector                    % R813
   '( [case_value_range,+] ')
   | 'default
end define
```

```
define case_value_range                % R814
   [case_value]
   | [case_value] ':
   | ': [case_value]
   | [case_value] ': [case_value]
end define

define case_value                      % R815
   [initialization_expr]
end define

% Construções DO. A sintaxe original exige o pareamento dos labels na
% instrução DO e na instrução terminal, mas não há como verificar isso em
% TXL durante a fase de análise léxica (parsing). A gramática foi
% simplificada e o DO foi considerado um [action_stmt] como os
% outros, ignorando o pareamento.

define do_stmt                         % R818
   [label_do_stmt]
   | [nonlabel_do_stmt]
end define

define label_do_stmt                   % R819
   'do [label] [loop_control?] [comment*] '; [NL]
end define

define nonlabel_do_stmt                % R820
   'do [loop_control?] [comment*] '; [NL]
end define

define loop_control                    % R821
   [comma?] [do_variable] '= [numeric_expr] [comma]
      [numeric_expr] [increment_part?]
   | [comma?] 'while '( [logical_expr] ')
end define

define increment_part
   [comma] [numeric_expr]
end define

define do_variable                     % R822
   [scalar_variable]
end define

define end_do_stmt                     % R825
   'end 'do [comment*] '; [NL]
end define

define cycle_stmt                      % R834
   'cycle [comment*] '; [NL]
end define

define exit_stmt                       % R835
   'exit [comment*] '; [NL]
end define

define goto_stmt                       % R836
   'go 'to [label] [comment*] '; [NL]
end define

define computed_goto_stmt              % R837
   'go 'to '( [label,+] ') [comma?] [int_expr] [comment*] '; [NL]
end define

define assign_stmt                     % R838
   'assign [label] 'to [int_variable] [comment*] '; [NL]
end define

define assigned_goto_stmt              % R839
   'go 'to [int_variable] [assigned_goto_part?] [comment*] '; [NL]
end define

define assigned_goto_part
   [comma?] '( [label,+] ')
end define
```

```
define arithmetic_if_stmt              % R840
   'if '( [numeric_expr] ') [label] ', [label] ', [label]
   [comment*] '; [NL]
end define

define continue_stmt                   % R841
   'continue [comment*] '; [NL]
end define

define stop_stmt                       % R842
   'stop [stop_code?] [comment*] '; [NL]
end define

define stop_code                       % R843
   [char_constant]
   |  [stop_int_code]
end define

define stop_int_code
   [integernumber]                     % one to five digits
end define

define pause_stmt                      % R844
   'pause [stop_code?] [comment*] '; [NL]
end define


% D.1.9 Input/Output Statements. ------------------------------------------

define io_unit                         % R901
   '*
   | [internal_file_unit]
   | [external_file_unit]
end define

define external_file_unit              % R902
   [int_expr]
end define

define internal_file_unit              % R903
   [char_variable]
end define

define open_stmt                       % R904
   'open '( [connect_spec,+] ') [comment*] '; [NL]
end define

define connect_spec                    % R905
   [unit_lead?] [external_file_unit]
   | 'iostat   '= [int_variable]
   | 'err      '= [label]
   | 'file     '= [file_name_expr]
   | 'status   '= [char_expr]
   | 'access   '= [char_expr]
   | 'form     '= [char_expr]
   | 'recl     '= [int_expr]
   | 'blank    '= [char_expr]
   | 'position '= [char_expr]
   | 'action   '= [char_expr]
   | 'delim    '= [char_expr]
   | 'pad      '= [char_expr]
   | 'carriagecontrol '= [char_expr]   % obs.: this is a non-ISO extension
end define

define file_name_expr                  % R906
   [char_expr]
end define

define close_stmt                      % R907
   'close '( [close_spec,+] ') [comment*] '; [NL]
end define

define close_spec                      % R908
   [unit_lead?] [external_file_unit]
   | 'iostat   '= [int_variable]
```

```
   | 'err      '= [label]
   | 'status   '= [char_expr]
end define

define read_stmt                        % R909
   'read '( [io_control_spec_part] ') [input_item,] [comment*] '; [NL]
   | 'read [format] [comment*] '; [NL]
   | 'read [format] [comma] [input_item,+] [comment*] '; [NL]
end define

define write_stmt                       % R910
   'write '( [io_control_spec_part] ') [output_item,] [comment*] '; [NL]
end define

define print_stmt                       % 911
   'print [format] [comment*] '; [NL]
   | 'print [format] [comma] [output_item,+] [comment*] '; [NL]
end define

define io_control_spec_part
   [io_unit] [io_control_extra_spec?]
   | [io_unit] [comma] [format] [io_control_extra_spec?]
   | [io_control_spec,+]
end define

define io_control_extra_spec
   [comma] [io_control_spec,+]
end define

define io_control_spec                  % R912
   [unit_lead] [io_unit]
   | 'fmt      '= [format]
   | 'nml      '= [namelist_group_name]
   | 'rec      '= [int_expr]
   | 'iostat   '= [int_variable]
   | 'err      '= [label]
   | 'end      '= [label]
   | 'advance  '= [char_expr]
   | 'size     '= [int_expr]
   | 'eor      '= [label]
end define

define unit_lead
   'unit '=
end define

define format                           % R913
   '*
   | [label]
   | [int_variable]
   | [char_expr]
end define

define input_item                       % R914
   [variable]
   | [io_implied_do]
end define

define output_item                      % R915
   [expr]
   | [io_implied_do]
end define

define io_implied_do                    % R916
   '( [io_implied_do_object,+] [comma] [io_implied_do_control] ')
end define

define io_implied_do_object             % R917
   [input_item]
   | [output_item]
end define

define io_implied_do_control            % R918
   [do_variable] '= [numeric_expr] [comma] [numeric_expr] [increment_part?]
end define
```

```
define backspace_stmt                   % R919
   'backspace [external_file_unit] [comment*] ';[NL]
   | 'backspace '( [position_spec,+] ')  [comment*] ';[NL]
end define

define endfile_stmt                     % R920
   'end 'file [external_file_unit] [comment*] '; [NL]
   | 'end 'file '( [position_spec,+] ') [comment*] '; [NL]
end define

define rewind_stmt                      % R921
   'rewind [external_file_unit] [comment*] '; [NL]
   | 'rewind '( [position_spec,+] ') [comment*] '; [NL]
end define

define position_spec                    % R922
   [unit_lead?] [external_file_unit]
   | 'iostat '= [int_variable]
   | 'err    '= [label]
end define

define inquire_stmt                     % R923
   'inquire '( [inquire_spec,+] ') [comment*] '; [NL]
end define

define inquire_spec                     % R924
   [unit_lead?] [external_file_unit]
   | 'file        '= [file_name_expr]
   | 'iostat      '= [int_variable]
   | 'err         '= [label]
   | 'exist       '= [logical_variable]
   | 'opened      '= [logical_variable]
   | 'number      '= [int_variable]
   | 'named       '= [logical_variable]
   | 'name        '= [char_expr]
   | 'access      '= [char_expr]
   | 'sequential  '= [char_expr]
   | 'direct      '= [char_expr]
   | 'form        '= [char_expr]
   | 'formatted   '= [char_expr]
   | 'unformatted '= [char_expr]
   | 'recl        '= [int_expr]
   | 'nextrec     '= [int_expr]
   | 'blank       '= [char_expr]
   | 'position    '= [char_expr]
   | 'action      '= [char_expr]
   | 'read        '= [char_expr]
   | 'write       '= [char_expr]
   | 'readwrite   '= [char_expr]
   | 'delim       '= [char_expr]
   | 'pad         '= [char_expr]
end define


% D.1.10 Input/Output Editing. ---------------------------------------------

define format_stmt                      % R1001
   [label] 'format '( [stringlit+] ') [comment*] '; [NL]
end define

% Como não há necessidade de manipular ou transformar [format_specification],
% a especificação de formato, transformada em uma seqüência de [stringlit]s
% pelo pré-processador, é reconhecida de uma tacada só.


% D.1.11 Program Units. ----------------------------------------------------

define main_program                     % R1101
   [comment*]
   [program_stmt?]
   [specification_part?]
   [execution_part?]
   [end_stmt]
end define

define program_stmt                     % R1102
```

```
   'program [program_name] '; [NL]
end define

define program_name
   [program_unit_name]
end define

define program_unit_name
   [id]
end define

define end_stmt                         % R1103
   'end [comment*] '; [NL]
end define

define block_data                       % R1110
   [comment*]
   [block_data_stmt]
   [specification_part?]
   [end_stmt]
end define

define block_data_stmt                  % R1111
   'block 'data [block_data_name?] '; [NL]
end define

define block_data_name
   [program_unit_name]
end define


% D.1.12 Procedures. ----------------------------------------------------

define external_stmt                    % R1207
   'external [external_name,+] [comment*] '; [NL]
end define

define external_name
   [id]
end define

define intrinsic_stmt                   % R1208
   'intrinsic [intrinsic_procedure_name,+] [comment*] '; [NL]
end define

define intrinsic_procedure_name
   [id]
end define

define function_reference               % R1209
   [function_name] [delim_arg_list]
end define

define delim_arg_list
   '( [actual_arg,] ')
end define

define function_name
   [id]
end define

define call_stmt                        % R1210
   'call [subroutine_name] [delim_arg_list?] [comment*] '; [NL]
end define

define subroutine_name
   [id]
end define

define actual_arg                       % R1213
   [expr]
   | [variable]
   | [alt_return_spec]
end define

define alt_return_spec                  % R1214
```

```
   '* [label]
end define

define function_subprogram             % R1215
   [comment*]
   [function_stmt]
   [specification_part?]
   [execution_part?]
   [end_stmt]
end define

define function_stmt                   % R1216
   [type_spec?] 'function [function_name] '( [dummy_arg_name,] '); [NL]
end define

define subroutine_subprogram           % R1219
   [comment*]
   [subroutine_stmt]
   [specification_part?]
   [execution_part?]
   [end_stmt]
end define

define subroutine_stmt                 % R1220
   'subroutine [subroutine_name] [delim_dummy_arg_list?] '; [NL]
end define

define delim_dummy_arg_list
   '( [dummy_arg,] ')
end define

define dummy_arg                       % R1221
   [dummy_arg_name]
   | '*
end define

define dummy_arg_name
   [id]
end define

define entry_stmt                      % R1223
   'entry [entry_name] [delim_dummy_arg_list?] '; [NL]
end define

define entry_name
   [id]
end define

define return_stmt                     % R1224
   'return [int_expr?] [comment*] '; [NL]
end define

define stmt_function_stmt              % R1226
   [function_name] '( [dummy_arg_name,] ') '= [expr] [comment*] '; [NL]
end define

define program
   [executable_program]
end define

% fim de for.grm
```

for_projreport.xslt

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--
   PUC-Rio - Departamento de Informática

   Arquivo:    for_projreport.xslt
   Projeto:    Lux
   Conteúdo:   Gerador de relatórios para fontes fortran
   Autor:      Marcelo Jaccoud Amaral
   Data:       2/5/2001

   Descrição:
      Esta transformação gera um documento XHTML 1.0 que
      recupera informações de um arquivo XMl contendo dados
      de um projeto (nome do projeto e arquivos fortran que o
      compõem). Um relatório é então gerado contendo um resumo
      das unidades de programa definidos em cada um dos arquivos.
      Esta folha de estilos ilustra como se pode manipular diversos
      fontes em cascata. Procedimentos semelhantes podem ser efetuados
      para realizar transformações que se propaguem em diversos
      arquivos, iterativamente ou recursivamente.
-->

<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:dc="http://purl.org/dc/elements/1.1"
   xmlns:lux="http://www.inf.puc-rio.br/lux/rdf/0.1"
   >
   <xsl:output
      method="xml"
      version="1.0"
      encoding="iso-8859-1"
      indent="yes"
      doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"
      doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
      />


<!--
   Gabarito principal.
   Cria os elementos XHTML principais, insere o título do projeto e
   monta o cabeçalho de uma tabela para conter os elementos.
-->
<xsl:template match="project">
   <html xmlns="http://www.w3.org/1999/xhtml">
      <head>
         <title>Relatório do projeto <xsl:value-of select="name"/></title>
      </head>
      <body>
         <h1>Projeto <xsl:value-of select="name"/></h1>
         <table border="solid" width="100%" bgcolor="beige">
            <caption>Relatório de componentes por arquivo</caption>
            <tr bgcolor="bisque">
               <th>Arquivo</th>
               <th>Unidade</th>
               <th>Tipo</th>
               <th>Tamanho<sup>*</sup></th>
               <th>Faz E/S?</th>
            </tr>
            <!-- processa os arquivos em ordem alfabética -->
            <xsl:apply-templates select="file">
               <xsl:sort/>
            </xsl:apply-templates>

         </table>
      <p><sup>*</sup>Número de subelementos XML na unidade.</p>
      </body>
   </html>
</xsl:template>


<!--
```

```
   Gabarito para processar um membro do projeto.
   O arquivo XMl correspondente ao código fortran é carregado, e
   algumas consultas são efetuadas na árvore, extraindo informações
   estruturais e textuais.
-->
<xsl:template match="file">
   <!-- nome do arquivo original -->
   <xsl:variable name="ofile" select="."/>
   <!-- nome do arquivo xml correspondente -->
   <xsl:variable name="xfile" select="concat(.,'.xml')"/>
   <!-- conteúdo do hiperfonte do membro -->
   <xsl:variable name="xdoc" select="document($xfile)"/>
   <!-- seleciona as unidades de programa contidas no fonte -->
   <xsl:variable name="units"
      select="$xdoc//subroutine | $xdoc//function |
              $xdoc//main-program | $xdoc//block-data"/>
   <!-- conta o número de elementos do arquivo -->
   <xsl:variable name="unit-count" select="count($units)"/>

   <!-- processa as unidades na ordem original do fonte -->
   <xsl:for-each select="$units">
      <tr>
         <xsl:if test="position()=1">
            <td rowspan="{$unit-count}"><xsl:value-of select="$ofile"/></td>
         </xsl:if>
         <td><xsl:value-of select="nm"/></td>
         <td><xsl:value-of select="local-name()"/></td>
         <td><xsl:value-of select="count(.//*)"/></td>
         <xsl:variable name="io-stmts"
            select="count($xdoc//write | $xdoc//read | $xdoc//open | $xdoc//close |
                          $xdoc//backspace | $xdoc//end-file | $xdoc//inquire |
                          $xdoc//rewind | $xdoc//print)"/>
         <xsl:choose>
            <xsl:when test="$io-stmts">
               <td bgcolor="#ffddcc">sim (<xsl:value-of select="$io-stmts"/>)</td>
            </xsl:when>
            <xsl:otherwise>
               <td bgcolor="#ccffcc">não</td>
            </xsl:otherwise>
         </xsl:choose>
      </tr>
   </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

cpp.dtd

```
<!ENTITY % block-declaration  "(simple-declaration | asm-definition |
              namespace-alias-definition | using-declaration | using-directive)" >

<!ENTITY % pp-if-line  "(pp-if | elif | ifdef | ifndef | else | endif)" >

<!ENTITY % pp-control-line  "(include-local | include-system | define-symbol |
              define-macro | undef | line | error | pragma | blank)" >

<!ENTITY % pp-directive  "(%pp-if-line; | %pp-control-line;)" >

<!ENTITY % declaration  "(%block-declaration; | function-definition |
              template-declaration | explicit-instatiation |
              explicit-specialization | linkage-specification |
              namespace-definition | %pp-directive;)" >

<!ENTITY % storage-class-specifier  "(auto | register | static | extern | mutable)" >

<!ENTITY % function-specifier  "(inline | virtual | explicit)" >

<!ENTITY % type-name  "(nm,template-spec?)" >

<!ENTITY % cv-qualifier  "(const | volatile)" >

<!ENTITY % selection-statement  "(if | switch)" >

<!ENTITY % iteration-statement  "(while | do-while | for)" >

<!ENTITY % jump-statement  "(break | continue | return | goto)" >

<!ENTITY % statement  "(labeled-statement | expression-statement |
              compound-statement | %selection-statement; | %iteration-statement; |
              %jump-statement; | %block-declaration; | try-block | %pp-if-line;)" >

<!ENTITY % preprocessing-token  "(nm | lit | op)" >

<!ENTITY % template-id  "(nm,template-spec)" >

<!ENTITY % access-specifier  "(private | protected | public)" >

<!ELEMENT translation-unit  (%declaration;)* >

<!ELEMENT function-definition  (decl-specifier*,function-declarator,((
              ctor-initializer?,function-body) | function-try-block)) >

<!ELEMENT template-declaration  (export?,template-parameter+,%declaration;) >

<!ELEMENT explicit-instatiation  %declaration; >

<!ELEMENT explicit-specialization  %declaration; >

<!ELEMENT linkage-specification  (lit,(%declaration;)*) >

<!ELEMENT namespace-definition  (nm?,(%declaration;)*) >

<!ELEMENT simple-declaration  (decl-specifier*,init-declarator*) >

<!ELEMENT asm-definition  (lit) >

<!ELEMENT namespace-alias-definition  (nm,global?,nested-name-specifier?,nm) >

<!ELEMENT using-declaration  ((nested-name-specifier | (global,
              nested-name-specifier?) | (typename,global?,nested-name-specifier)),
              unqualified-id) >

<!ELEMENT using-directive  (global?,nested-name-specifier?,nm) >

<!ELEMENT decl-specifier  (%storage-class-specifier; | %function-specifier; |
              friend | typedef | type-specifier) >

<!ELEMENT function-declarator  (ptr-operator*,direct-declarator,function-parcel) >

<!ELEMENT ctor-initializer  (mem-initializer+) >
```

```
<!ELEMENT function-body  (compound-statement) >

<!ELEMENT function-try-block  (ctor-initializer?,function-body,handler+) >

<!ELEMENT export   EMPTY  >

<!ELEMENT template-parameter  (type-parameter | parameter-declaration) >

<!ELEMENT lit  (#PCDATA) >

<!ELEMENT nm  (#PCDATA) >

<!ELEMENT init-declarator  (declarator,initializer?) >

<!--global namespace specifier (prefixed ::) -->
<!ELEMENT global   EMPTY  >

<!ELEMENT nested-name-specifier  (%type-name;,(template?,%type-name;)*) >

<!ELEMENT typename   EMPTY  >

<!ELEMENT unqualified-id  (operator-function-id | conversion-function-id | (tilde?,
              %type-name;)) >

<!ELEMENT friend   EMPTY  >

<!ELEMENT typedef   EMPTY  >

<!ELEMENT type-specifier  (simple-type-specifier | class-specifier |
              %cv-qualifier; | enum-specifier | elaborated-type-specifier) >

<!ELEMENT ptr-operator  ((global?,nested-name-specifier)?,op,(%cv-qualifier;)*) >

<!ELEMENT direct-declarator  ((declarator-id | declarator),(vector-parcel |
              function-parcel)*) >

<!ELEMENT function-parcel  (parameter-declaration-clause,(%cv-qualifier;)*,
              exception-specification?) >

<!ELEMENT mem-initializer  (global?,nested-name-specifier?,%type-name;,expression?) >

<!ELEMENT compound-statement  (%statement;)* >

<!ELEMENT handler  (exception-declaration,compound-statement) >

<!ELEMENT type-parameter  (((typename | class),nm?,type-id?) | (template-parameter+,
              nm?,id-expression)) >

<!ELEMENT parameter-declaration  (decl-specifier+,(declarator | abstract-declarator)?,
              assignment-expression?) >

<!ELEMENT pp-if  (conditional-expression) >

<!ELEMENT elif  (conditional-expression) >

<!ELEMENT ifdef  (nm) >

<!ELEMENT ifndef  (nm) >

<!ELEMENT else   EMPTY  >

<!ELEMENT endif   EMPTY  >

<!ELEMENT include-local  (%preprocessing-token;)+ >

<!ELEMENT include-system  (%preprocessing-token;)+ >

<!ELEMENT define-symbol  (nm,replacement) >

<!ELEMENT define-macro  (nm+,replacement) >

<!ELEMENT undef  (nm) >

<!ELEMENT line  (%preprocessing-token;)+ >
```

```
<!ELEMENT error  (%preprocessing-token;)+ >

<!ELEMENT pragma  (%preprocessing-token;)+ >

<!ELEMENT blank   EMPTY  >

<!ELEMENT declarator  (ptr-operator*,direct-declarator) >

<!ELEMENT initializer  (initializer-clause | expression) >

<!ELEMENT template   EMPTY  >

<!ELEMENT operator-function-id  (op) >

<!ELEMENT conversion-function-id  (type-specifier+,ptr-operator*) >

<!ELEMENT tilde   EMPTY  >

<!ELEMENT auto   EMPTY  >

<!ELEMENT register   EMPTY  >

<!ELEMENT static   EMPTY  >

<!ELEMENT extern   EMPTY  >

<!ELEMENT mutable   EMPTY  >

<!ELEMENT inline   EMPTY  >

<!ELEMENT virtual   EMPTY  >

<!ELEMENT explicit   EMPTY  >

<!ELEMENT simple-type-specifier  ((global?,(%type-name; | (nested-name-specifier,(
              %type-name; | (template,%template-id;))))) | char | wchar_t | bool |
              short | int | long | signed | unsigned | float | double | void) >

<!ELEMENT class-specifier  (class-head,(%access-specifier; | member-declaration |
              %pp-if-line;)*) >

<!ELEMENT enum-specifier  (nm?,enumerator-definition*) >

<!ELEMENT elaborated-type-specifier  (((class | struct | union | enum),global?,
              nested-name-specifier?,nm) | (typename,global?,nested-name-specifier,(
              %type-name; | (template,%template-id;)))) >

<!ELEMENT op  (#PCDATA) >

<!ELEMENT declarator-id  (id-expression | (global?,nested-name-specifier?,
              %type-name;)) >

<!ELEMENT vector-parcel  (conditional-expression?) >

<!ELEMENT parameter-declaration-clause  (parameter-declaration*,ellipsis?) >

<!ELEMENT exception-specification  (type-id*) >

<!ELEMENT expression  (assignment-expression+) >

<!ELEMENT exception-declaration  ((type-specifier+,(declarator |
              abstract-declarator)?) | ellipsis) >

<!ELEMENT class   EMPTY  >

<!ELEMENT type-id  (type-specifier+,abstract-declarator?) >

<!ELEMENT id-expression  (unqualified-id | qualified-id) >

<!ELEMENT abstract-declarator  ((ptr-operator+,direct-abstract-declarator?) |
              direct-abstract-declarator) >

<!ELEMENT assignment-expression  (conditional-expression | throw-expression | (
              binary-expression,op,assignment-expression)) >

<!ELEMENT conditional-expression  (binary-expression,(expression,
```

```
              assignment-expression)?) >

<!ELEMENT replacement  (%preprocessing-token;)* >

<!ELEMENT initializer-clause  (assignment-expression | aggregate) >

<!ELEMENT template-spec  (template-argument*) >

<!ELEMENT char   EMPTY  >

<!ELEMENT wchar_t   EMPTY  >

<!ELEMENT bool   EMPTY  >

<!ELEMENT short   EMPTY  >

<!ELEMENT int   EMPTY  >

<!ELEMENT long   EMPTY  >

<!ELEMENT signed   EMPTY  >

<!ELEMENT unsigned   EMPTY  >

<!ELEMENT float   EMPTY  >

<!ELEMENT double   EMPTY  >

<!ELEMENT void   EMPTY  >

<!ELEMENT class-head  ((class | struct | union),(nested-name-specifier?,
              %type-name;)?,base-specifier*) >

<!ELEMENT member-declaration  ((global?,nested-name-specifier,template?,
              unqualified-id) | (decl-specifier*,member-declarator*) |
              function-definition | using-declaration | template-declaration) >

<!ELEMENT const   EMPTY  >

<!ELEMENT volatile   EMPTY  >

<!ELEMENT enumerator-definition  (nm,conditional-expression?) >

<!ELEMENT struct   EMPTY  >

<!ELEMENT union   EMPTY  >

<!ELEMENT enum   EMPTY  >

<!ELEMENT ellipsis   EMPTY  >

<!ELEMENT labeled-statement  ((nm | (case,conditional-expression) | default),
              %statement;) >

<!ELEMENT expression-statement  (expression?) >

<!ELEMENT try-block  (compound-statement,handler+) >

<!ELEMENT qualified-id  ((nested-name-specifier,template?,unqualified-id) | (global,((
              nested-name-specifier,template?,unqualified-id) |
              operator-function-id | %type-name;))) >

<!ELEMENT direct-abstract-declarator  ((abstract-declarator | vector-parcel |
              function-parcel),(vector-parcel | function-parcel)*) >

<!ELEMENT throw-expression  (assignment-expression?) >

<!ELEMENT binary-expression  (cast-expression,(op | cast-expression)*) >

<!ELEMENT aggregate  (assignment-expression | aggregate)* >

<!ELEMENT template-argument  (assignment-expression | type-id | id-expression) >

<!ELEMENT base-specifier  (virtual?,(%access-specifier;)?,global?,
              nested-name-specifier?,%type-name;) >
```

```
<!ELEMENT private   EMPTY  >

<!ELEMENT protected   EMPTY  >

<!ELEMENT public   EMPTY  >

<!ELEMENT member-declarator  ((nm?,conditional-expression) | (declarator,(
              pure-specifier | conditional-expression)?)) >

<!ELEMENT case   EMPTY  >

<!ELEMENT default   EMPTY  >

<!ELEMENT if  (condition,%statement;,(%statement;)?) >

<!ELEMENT switch  (condition,%statement;) >

<!ELEMENT while  (condition,%statement;) >

<!ELEMENT do-while  (%statement;,expression) >

<!ELEMENT for  (for-init,condition?,expression?,%statement;) >

<!ELEMENT break   EMPTY  >

<!ELEMENT continue   EMPTY  >

<!ELEMENT return  (expression?) >

<!ELEMENT goto  (nm) >

<!ELEMENT cast-expression  ((type-id,cast-expression) | unary-expression) >

<!ELEMENT pure-specifier   EMPTY  >

<!ELEMENT condition  (expression | (type-specifier+,declarator,
              assignment-expression)) >

<!ELEMENT for-init  (expression-statement | simple-declaration) >

<!ELEMENT unary-expression  (postfix-expression | (op,(cast-expression |
              unary-expression | type-id)) | new-expression | delete-expression) >

<!ELEMENT postfix-expression  (postfix-operand,postfix-parcel*) >

<!ELEMENT new-expression  (global?,new-placement?,(new-type-id | type-id),
              new-initializer?) >

<!ELEMENT delete-expression  (global?,op?,cast-expression) >

<!ELEMENT postfix-operand  (primary-expression | typename-cast | dynamic-cast |
              static-cast | reinterpret-cast | const-cast | simple-type-cast) >

<!ELEMENT postfix-parcel  (array-index | function-call | (op,(
              pseudo-destructor-name | (template?,id-expression))?)) >

<!ELEMENT new-placement  (expression) >

<!ELEMENT new-type-id  (type-specifier+,new-declarator?) >

<!ELEMENT new-initializer  (expression?) >

<!ELEMENT primary-expression  (lit | this | expression | id-expression) >

<!ELEMENT typename-cast  (global?,nested-name-specifier,(%type-name; | (template,
              %template-id;)),expression?) >

<!ELEMENT dynamic-cast  (type-id,expression) >

<!ELEMENT static-cast  (type-id,expression) >

<!ELEMENT reinterpret-cast  (type-id,expression) >

<!ELEMENT const-cast  (type-id,expression) >

<!ELEMENT simple-type-cast  (simple-type-specifier,expression?) >
```

```
<!ELEMENT array-index  (expression) >

<!ELEMENT function-call  (expression?) >

<!ELEMENT pseudo-destructor-name  (global?,((%type-name;,(%type-name;)?) | (
              nested-name-specifier,((%type-name;,(%type-name;)?) | (template,
              %template-id;,%type-name;))))) >

<!ELEMENT new-declarator  (direct-new-declarator | (ptr-operator+,
              direct-new-declarator?)) >

<!ELEMENT this   EMPTY  >

<!ELEMENT direct-new-declarator  (expression,conditional-expression*) >

<!ATTLIST asm-definition
              id  ID    #IMPLIED  >

<!ATTLIST break
              id  ID    #IMPLIED  >

<!ATTLIST class-specifier
              id  ID    #IMPLIED  >

<!ATTLIST compound-statement
              id  ID    #IMPLIED  >

<!ATTLIST continue
              id  ID    #IMPLIED  >

<!ATTLIST define-macro
              id  ID    #IMPLIED  >

<!ATTLIST define-symbol
              id  ID    #IMPLIED  >

<!ATTLIST do-while
              id  ID    #IMPLIED  >

<!ATTLIST elif
              id  ID    #IMPLIED  >

<!ATTLIST else
              id  ID    #IMPLIED  >

<!ATTLIST endif
              id  ID    #IMPLIED  >

<!ATTLIST error
              id  ID    #IMPLIED  >

<!ATTLIST explicit-instatiation
              id  ID    #IMPLIED  >

<!ATTLIST explicit-specialization
              id  ID    #IMPLIED  >

<!ATTLIST expression-statement
              id  ID    #IMPLIED  >

<!ATTLIST for
              id  ID    #IMPLIED  >

<!ATTLIST function-definition
              id  ID    #IMPLIED  >

<!ATTLIST goto
              id  ID    #IMPLIED  >

<!ATTLIST handler
              id  ID    #IMPLIED  >

<!ATTLIST if
              id  ID    #IMPLIED  >
```

```
<!ATTLIST ifdef
              id  ID    #IMPLIED  >

<!ATTLIST ifndef
              id  ID    #IMPLIED  >

<!ATTLIST include-local
              id  ID    #IMPLIED  >

<!ATTLIST include-system
              id  ID    #IMPLIED  >

<!ATTLIST init-declarator
              id  ID    #IMPLIED  >

<!ATTLIST labeled-statement
              id  ID    #IMPLIED  >

<!ATTLIST line
              id  ID    #IMPLIED  >

<!ATTLIST linkage-specification
              block (true | false)  "false"
              id  ID    #IMPLIED  >

<!ATTLIST lit
              t (integer | character | string | floating | boolean | number)
               #IMPLIED
              mod  CDATA    #IMPLIED
              xml:space (preserve | default)  "default" >

<!ATTLIST member-declaration
              id  ID    #IMPLIED  >

<!ATTLIST member-declarator
              id  ID    #IMPLIED  >

<!ATTLIST namespace-alias-definition
              id  ID    #IMPLIED  >

<!ATTLIST namespace-definition
              id  ID    #IMPLIED  >

<!ATTLIST nm
              type  CDATA    #IMPLIED
              cat (template | class | enum | namespace | typedef | label |
              enumerator | namespace-alias | pp-symbol)   #IMPLIED  >

<!ATTLIST parameter-declaration
              id  ID    #IMPLIED  >

<!ATTLIST postfix-expression
              op  CDATA    #IMPLIED  >

<!ATTLIST pp-if
              id  ID    #IMPLIED  >

<!ATTLIST pragma
              id  ID    #IMPLIED  >

<!ATTLIST pseudo-destructor-name
              template (true | false)  "false" >

<!ATTLIST return
              id  ID    #IMPLIED  >

<!ATTLIST simple-declaration
              id  ID    #IMPLIED  >

<!ATTLIST simple-type-specifier
              template (true | false)  "false" >

<!ATTLIST switch
              id  ID    #IMPLIED  >

<!ATTLIST template-declaration
```

```
              id  ID    #IMPLIED  >

<!ATTLIST translation-unit
              id  ID    #IMPLIED
              source  CDATA    #IMPLIED  >

<!ATTLIST try-block
              id  ID    #IMPLIED  >

<!ATTLIST undef
              id  ID    #IMPLIED  >

<!ATTLIST using-declaration
              id  ID    #IMPLIED  >

<!ATTLIST using-directive
              id  ID    #IMPLIED  >

<!ATTLIST while
              id  ID    #IMPLIED  >
```

cpp.grm

```
% PUC-Rio - Departamento de Informática
%=============================================================================
%
%  File name:     isocpp.grm
%  Project:       Lux
%  Content:       C++ Grammar according to ISO/IEC 14882:1998
%  Author:        Marcelo Jaccoud Amaral
%  Date:          02/05/2001
%
% Description:
%  C++ grammar tailored for translation of ISO-conforming code into Lux.
%  Departures result from:
%  (a) TXLs inability to handle some constructs correctly, like for example
%     the escape sequence for Unicode characters in identifiers;
%  (b) the content preservation directive adopted in the translation to Lux;
%     for exemple, preprocessor commands are transliterated rather then
%     expanded.
%  Nevertheless, the grammar was written to accept a superset of the valid
%  C++ source. For example, due to line size limitations (TXL cannot process
%  lines longer then 254 characters) the correct expression for the [id]
%  literal, which includes the universal-character-name escape sequence (\u),
%  cannot be defined. The solution was to include '\' as a valid id char.
%  Although this means accepting non-valid names as m\ud (only 4 or 8 hex
%  digits are allowed), this guarantees that all valid names will parse
%  correctly. Removing support for \u sequences would hinder the translation
%  of correct code.
%
%  The guidelines were:
%     - must parse all valid ISO-C++ code;
%     - may parse code that is invalid but looks lexically correct;
%
%  Restriction:
%     - Comment position is limited to reasonable locations.
%     - Since C++ is context sensitive, some declarations may be parsed
%        to the wrong structures. For example, "a f(b);" may be either a
%        function f receiving an argument of type b and returning a object
%        of type a or a declaration of a object f of class a initialized with
%        a variable named b. To resolve the ambiguity we need to know if b
%        is a type name or a variable name, and TXL has no way of handling
%        such cases.
%
% Reference:
%  Almost all syntax rules have been transliterated from the ISO standard.
%  The same names and constructions have been used, and the sections were
%  numbered like the related sections in the standard, to ease navigation.
%  These are placed at the right side, where suitable, e.g. [lex.key].
%=============================================================================

%=================== These definitions are TXL-related =====================

% Comments are considered significant, and their positions are only valid
% where allowed by the syntax. Allowing comments to appear anywhere is not
% viable, so only the natural placements were considered valid. Since it is
% very rare for comments to appear inside complex instructions, this is not
% so restrictive as it seems. Such strange placements are sometimes used to
% deactivate some portion of the code with a bounded comment /*...*/ ; in
% this case, replace the section with a corresponding #if (0) ... #endif.


comments                                                    %  [lex.comment]
   /*  */
   //
end comments

%                                                           [lex.trigraph]
% trigraphs    these where replaced in the preprocessor, since it would be
%              impossible to include them in the syntax
%
%     trigraph replacement
%     ??=      #
%     ??/      \
%     ??'      ^
```

```
%     ??(      [
%     ??)      ]
%     ??!      |
%     ??<      {
%     ??>      }
%     ??-      ~


%                                                              [lex.digraph]
%  Alternative tokens
%  Since the alternative tokens can be used wherever the primaries can,
%  they should be defined in common nonterminals.
%  However, these are rarely used nowadays, and posses significant parsing
%  difficulties in TXL because of the percent characters.
%  They where left for a future version.
%
%      altern.  primary
%      <%       {
%      %>       }
%      <:       [
%      :>       ]
%      %:       #
%      %:%:     ##
%      and      &&
%      bitor    |
%      or       ||
%      xor      ^
%      compl    ~
%      bitand   &
%      and_eq   &=
%      or_eq    |=
%      xor_eq   ^=
%      not      !
%      not_eq   !=
%


compounds                               %  [lex.digraph][lex.operators]
   '##
   '...
   '+= '-= '*= '/=
   '%=
   '^= '&= '|=
   '<< '>> '<<= '>>=
   '== '!= '<= '>= '&&
   '||
   '++ '-- '->* '->
   '.* '::
   '~>                    % special digraph inserted by the code morpher to
                          % mark the ending of a preprocessind directive
end compounds

% Literal alternative tokens (such as 'and') where left in the keys
% definitions to enforce ISO conformance (they are reserved keywords).
% However, they were not implemented. (see note above)

keys                                    %  [lex.key][lex.digraph]
   'and 'and_eq asm auto
   bitand bitor bool break
   case catch char class compl const const_cast continue
   default do double dynamic_cast
   else enum explicit export extern
   false float for friend
   if inline int
   long
   mutable
   namespace new 'not not_eq
   operator or or_eq
   private protected public
   register return
   short signed sizeof static static_cast struct switch
   template this throw true typedef typeid typename
   union unsigned using
   virtual void volatile
   wchar_t while
   xor xor_eq
```

```
end keys

%============================================================================
% A.1 Keywords                                                    [gram.key]
%============================================================================

define type_name
   [id] [template_spec?]
end define

define template_id
   [id] [template_spec]
end define

define template_spec
   '< [template_argument,] '>
end define


%============================================================================
% A.2 Lexical conventions                                         [gram.lex]
%============================================================================

% The following pragma is needed to allow hex-quads in a name. Correctly
% defining the id token would require a regular expression 305 characteres
% long, which TXL refuses to compile because of the 254-characters-per-line
% limit. This pragma will also allow invalid escape sequences, but since the
% backslash character has no other function in the sintax, it will not
% affect the parsing of correct code.
% The correct expression should be (ignoring the spaces and line breaks):
%
% [\t(\\[uU][\dabcdefABCDEF][\dabcdefABCDEF][\dabcdefABCDEF][\dabcdefABCDEF]
% (\\[uU][\dabcdefABCDEF][\dabcdefABCDEF][\dabcdefABCDEF][\dabcdefABCDEF])?)]
% [\t\d(\\[uU][\dabcdefABCDEF][\dabcdefABCDEF][\dabcdefABCDEF][\dabcdefABCDEF]
% (\\[uU][\dabcdefABCDEF][\dabcdefABCDEF][\dabcdefABCDEF][\dabcdefABCDEF])?)]*
%
%# pragma -idchars '\' -comment -w 255

define preprocessing_token                              % [lex.pptoken]
   [id]
   | [number]
   | [character_literal]
   | [string_literal]
   | [preprocessing_op_or_punc]
   | [key]
   % this line, present in the TXL C grammar, would undesirebly
   % catch the ~> special token and was thus removed
end define

define preprocessing_op_or_punc
   '{
   | '}
   | '[
   | ']
   | '#
   | '##
   | '(
   | ')
   | ';
   | ':
   | '...
   | 'new
   | 'delete
   | '?
   | '::
   | '.
   | '.*
   | '+
   | '-
   | '*
   | '/
   | '%
   | '^
   | '&
   | '|
   | '~
```

```
   | '!
   | '=
   | '<
   | '>
   | '+=
   | '-=
   | '*=
   | '/=
   | '%=
   | '^=
   | '&=
   | '|=
   | '<<
   | '>>
   | '>>=
   | '<<=
   | '==
   | '!=
   | '<=
   | '>=
   | '&&
   | '||
   | '++
   | '--
   | ',
   | '->*
   | '->
end define

%
% These tokens are defined in order to parse the C++ code, but they clash
% with TXL predefined number definitions.
% Specially problematic is the definition of number, which does not
% allow us to distinguish the C++ literals correctly. We redefined
% number as a make-up solution. The other tokens are used only when the
% TXL number does not get it.
%
tokens
   number               "\d*"
   decimal_literal      "[123456789]\d*[([uU][lL]?)([lL][uU]?)]"
   octal_literal        "0[01234567]*[([uU][lL]?)([lL][uU]?)]"
   hexadecimal_literal  "0[xX][0123456789AaBbCcDdEeFf]+[([uU][lL]?)([lL][uU]?)]?"
   floating_literal     "[(\d*.\d+)(\d+.)]([eE][+-]?\d+)?[flFL]"
end tokens

define literal
   [integer_literal]
   | [character_literal]
   | [floating_literal]
   | [string_literal]
   | [boolean_literal]
end define

define integer_literal
   [number]
   | [decimal_literal]
   | [octal_literal]
   | [hexadecimal_literal]
end define

define character_literal
   ['L ?] [charlit]
end define

define string_literal
   ['L ?] [stringlit+]
end define

define boolean_literal
   'false | 'true
end define

%============================================================================
% A.3 Basic concepts                                              [gram.basic]
%============================================================================
```

```
define translation_unit
   [comment*]
   [declaration*]
end define

%=============================================================================
% A.4 Expressions                                                  [gram.expr]
%=============================================================================

define primary_expression
   [literal]
   | 'this
   | '( [expression] ')
   | [id_expression]
end define

define id_expression
   [unqualified_id]
   | [qualified_id]
end define

define unqualified_id
   [type_name]
   | [operator_function_id]
   | [conversion_function_id]
   | '~ [type_name]
end define

define qualified_id
   ['':: ?] [nested_name_specifier] ['template ?] [unqualified_id]
   | '':: [type_name]
   | '':: [operator_function_id]
end define

define nested_name_specifier
   [type_name] '':: [nested_name_specifier?]
   | [type_name] '':: 'template [nested_name_specifier]
end define

define postfix_expression
   [postfix_operand] [postfix_parcel*]
end define

define postfix_operand
   [primary_expression]
   | 'typename ['':: ?] [nested_name_specifier] [typename_rest]
   | 'dynamic_cast '< [type_id] '> '( [expression] ')
   | 'static_cast '< [type_id] '> '( [expression] ')
   | 'reinterpret_cast '< [type_id] '> '( [expression] ')
   | 'const_cast '< [type_id] '> '( [expression] ')
   | [simple_type_specifier] '( [expression?] ')
end define

define typename_rest
   [id] '( [expression?] ')
   | ['template ?] [template_id] '( [expression?] ')
end define

define casted_type
   [expression]
   | [type_id]
end define

define postfix_parcel
   '[ [expression] ']
   | '( [expression?] ')
   | '. ['template ?] [id_expression]
   | '-> ['template ?] [id_expression]
   | '. [pseudo_destructor_name]
   | '-> [pseudo_destructor_name]
   | '++
   | '--
end define

define pseudo_destructor_name
   ['':: ?] [nested_name_specifier?] [type_name] '':: '~ [type_name]
```

```
   | ['::  ?] [nested_name_specifier] 'template [template_id] '::  '~ [type_name]
   | ['::  ?] [nested_name_specifier?] '~ [type_name]
end define

define unary_expression
   [postfix_expression]
   | '++ [cast_expression]
   | '-- [cast_expression]
   | [unary_operator] [cast_expression]
   | 'sizeof [unary_expression]
   | 'sizeof '( [type_id] ')
   | [new_expression]
   | [delete_expression]
end define

define unary_operator
   '*
   | '&
   | '+
   | '-
   | '!
   | '~
end define

define new_expression
   ['::  ?] 'new [new_placement?] [new_type_id] [new_initializer?]
   | ['::  ?] 'new [new_placement?] '( [type_id] ') [new_initializer?]
end define

define new_placement
   '( [expression] ')
end define

define new_type_id
   [type_specifier+] [new_declarator?]
end define

define new_declarator
   [ptr_operator+] [direct_new_declarator?]
   | [direct_new_declarator]
end define

define direct_new_declarator
   '[ [expression] '] [direct_new_parcel*]
end define

define direct_new_parcel
   '[ [conditional_expression] ']
end define

define new_initializer
   '( [expression?] ')
end define

define delete_expression
   ['::  ?] 'delete [delete_parcel]
end define

define delete_parcel
   [cast_expression]
   | '[ '] [cast_expression]
end define

define cast_expression
   [unary_expression]
   | '( [type_id] ') [cast_expression]
end define

define binary_operator
   '.*
   | '->*
   | '*
   | '/
   | '%
   | '+
   | '-
```

```
   | '<<
   | '>>
   | '<
   | '>
   | '<=
   | '>=
   | '==
   | '!=
   | '&
   | '^
   | '|
   | '&&
   | '||
end define

define binary_expression
   [cast_expression] [binary_parcel*]
end define

define binary_parcel
   [binary_operator] [cast_expression]
end define

define conditional_expression
   [binary_expression] [conditional_parcel?]
end define

define conditional_parcel
   '? [expression] : [assignment_expression]
end define

define assignment_expression
   [conditional_expression]
   | [binary_expression] [assignment_operator] [assignment_expression]
   | [throw_expression]
end define

define assignment_operator
   '=
   | '*=
   | '/=
   | '%=
   | '+=
   | '-=
   | '>>=
   | '<<=
   | '&=
   | '^=
   | '|=
end define

define expression
   [assignment_expression,+]
end define

%=============================================================================
% A.5 Statements                                                [gram.stmt.stmt]
%=============================================================================

define statement
   [labeled_statement]
   | [expression_statement]
   | [compound_statement]
   | [selection_statement]
   | [iteration_statement]
   | [jump_statement]
   | [declaration_statement]
   | [try_block]
%   % this was added to allow for preprocessing directives to be included
%   % in the syntax in a limited way
   | [if_line]
end define

define labeled_statement
   [id] ': [comment*] [statement]
   | 'case [conditional_expression] ': [comment*] [statement]
```

```
   | 'default ': [comment*] [statement]
end define

define expression_statement
   [expression?] '; [comment*]
end define

define compound_statement
   '{ [comment*] [statement*] '} [comment*]
end define

define selection_statement
   'if '( [condition] ') [comment*] [statement] [else_part?]
   | 'switch '( [condition] ') [comment*] [statement]
end define

define else_part
   'else [comment*] [statement]
end define

define condition
   [expression]
   | [type_specifier+] [declarator] '= [assignment_expression]
end define

define iteration_statement
   'while '( [condition] ') [comment*] [statement]
   | 'do [comment*] [statement] 'while '( [expression] ') '; [comment*]
   | 'for '( [for_init_statement] [condition?] '; [comment*] [expression?]
         ') [comment*] [statement]
end define

define for_init_statement
   [expression_statement]
   | [simple_declaration]
end define

define jump_statement
   'break '; [comment*]
   | 'continue '; [comment*]
   | 'return  [expression?] '; [comment*]
   | 'goto [id] '; [comment*]
end define

define declaration_statement
   [block_declaration]
end define

%=============================================================================
% A.6 Declarations                                              [gram.dcl.dcl]
%=============================================================================

define declaration
   [wide_block_declaration]                                          [NL]
   | [function_definition]                                           [NL]
   | [template_declaration]                                          [NL]
   | [explicit_instantiation]                                        [NL]
   | [explicit_specialization]                                       [NL]
   | [linkage_specification]                                         [NL]
   | [namespace_definition]                                          [NL]
%   % the following incorporates the preprocesing directives into the
%   % syntax in a limited way
   | [pp_directive]                                                  [NL]
end define

define block_declaration
   [simple_declaration]
   | [asm_definition]
   | [namespace_alias_definition]
   | [using_declaration]
   | [using_directive]
end define

define wide_block_declaration
   [wide_simple_declaration]
   | [asm_definition]
```

```
   | [namespace_alias_definition]
   | [using_declaration]
   | [using_directive]
end define

define simple_declaration
   [init_declarator,] '; [comment*]
   | [decl_specifier] [simple_declaration]
end define

define wide_simple_declaration
   [wide_init_declarator,] '; [comment*]
   | [decl_specifier] [wide_simple_declaration]
end define

define decl_specifier
   [storage_class_specifier]
   | [type_specifier]
   | [function_specifier]
   | 'friend
   | 'typedef
end define

define storage_class_specifier
   'auto
   | 'register
   | 'static
   | 'extern
   | 'mutable
end define

define function_specifier
   'inline
   | 'virtual
   | 'explicit
end define

define type_specifier
   [simple_type_specifier]
   | [class_specifier]
   | [enum_specifier]
   | [elaborated_type_specifier]
   | [cv_qualifier]
end define

define simple_type_specifier
   ['::  ?] [nested_name_specifier?] [type_name]
   | ['::  ?] [nested_name_specifier] 'template [template_id]
   | 'char
   | 'wchar_t
   | 'bool
   | 'short
   | 'int
   | 'long
   | 'signed
   | 'unsigned
   | 'float
   | 'double
   | 'void
end define

define elaborated_type_specifier
   [class_key] ['::  ?] [nested_name_specifier?] [id]
   | 'enum ['::  ?] [nested_name_specifier?] [id]
   | 'typename ['::  ?] [nested_name_specifier] [id]
   | 'typename ['::  ?] [nested_name_specifier] ['template ?] [template_id]
end define

define enum_specifier
   'enum [id?] [comment*] '{ [enumerator_definition,] '}
end define

define enumerator_definition
   [comment*] [enumerator] [comment*]
   | [comment*] [enumerator] '= [conditional_expression] [comment*]
end define
```

```
define enumerator
   [id]
end define

define namespace_definition
   [named_namespace_definition]
   | [unnamed_namespace_definition]
end define

define named_namespace_definition
   'namespace [id] [comment*] '{ [comment*] [namespace_body] '} [comment*]
end define

define unnamed_namespace_definition
   'namespace [comment*] '{ [comment*] [namespace_body] '} [comment*]
end define

define namespace_body
   [declaration*]
end define

define namespace_alias_definition
   'namespace [id] '= [qualified_namespace_specifier] '; [comment*]
end define

define qualified_namespace_specifier
   [':: ?] [nested_name_specifier?] [id]
end define

define using_declaration
   'using ['typename ?] ['::  ?] [nested_name_specifier]
         [unqualified_id] '; [comment*]
   | 'using '::  [unqualified_id] ; [comment*]
end define

define using_directive
   'using 'namespace  ['::  ?] [nested_name_specifier?] [id] '; [comment*]
end define

define asm_definition
   'asm [comment*] '( [comment*] [string_literal] [comment*] ') '; [comment*]
end define

define linkage_specification
   'extern [string_literal] [comment*] '{ [comment*] [declaration*]
                                       '} [comment*]
   | 'extern [string_literal] [declaration]
end define

%=============================================================================
% A.7 Declarators                                              [gram.dcl.decl]
%=============================================================================

define init_declarator
   [declarator] [initializer?]
end define

define wide_init_declarator
   [wide_declarator] [initializer?]
end define

define declarator
   [ptr_operator*] [direct_declarator]
end define

define wide_declarator
   [ptr_operator*] [wide_direct_declarator]
end define

define direct_declarator
   [declarator_id]
   | '( [declarator] ')
   | [wide_direct_declarator] [vector_parcel]
end define
```

```
define wide_direct_declarator
   [declarator_id] [declarator_parcel*]
   | '( [declarator] ')  [declarator_parcel*]
end define

define declarator_parcel
   [function_parcel]
   | [vector_parcel]
end define

define vector_parcel
   '[ [conditional_expression?] ']
end define

define function_parcel
   '( [parameter_declaration_clause] ') [cv_qualifier*] [exception_specification?]
end define

define function_declarator
   [ptr_operator*] [direct_declarator] [function_parcel]
end define

define ptr_operator
   '* [cv_qualifier*]
   | '&
   | [':: ?] [nested_name_specifier] '* [cv_qualifier*]
end define

define cv_qualifier
   'const
   | 'volatile
end define

define declarator_id
   [id_expression]
   | [':: ?] [nested_name_specifier?] [type_name]
end define

define type_id
   [type_specifier+] [abstract_declarator?]
end define

define abstract_declarator
   [ptr_operator+] [direct_abstract_declarator?]
   | [direct_abstract_declarator]
end define

define direct_abstract_declarator
   [direct_abstract_declarator_start] [declarator_parcel*]
end define

define direct_abstract_declarator_start
   '( [abstract_declarator] ')
   | [declarator_parcel]
end define

define parameter_declaration_clause
   [parameter_declaration,] ['... ?]
   | [parameter_declaration,+] ', '...
end define

define parameter_declaration
   [comment*] [decl_specifier+] [declarator] [comment*]
   | [comment*] [decl_specifier+] [declarator]
                '= [assignment_expression] [comment*]
   | [comment*] [decl_specifier+] [abstract_declarator?] [comment*]
   | [comment*] [decl_specifier+] [abstract_declarator?]
                '= [assignment_expression] [comment*]
end define

define function_definition
   [function_declarator] [comment*] [function_body_part]
   | [decl_specifier] [function_definition]
end define

define function_body_part
```

```
   [ctor_initializer?] [comment*] [function_body]
   | [function_try_block]
end define

define function_body
   [compound_statement]
end define

define initializer
   '= [initializer_clause]
   | '( [expression] ')
end define

define initializer_clause
   [assignment_expression]
   | '{ '}
   | '{ [initializer_clause,+] [', ?] '}
end define

%=============================================================================
% A.8 Classes                                                    [gram.class]
%=============================================================================

define class_specifier
   [class_head] [comment*] '{ [comment*] [member_specification*] '} [comment*]
end define

define class_head
   [class_key] [class_head_rest?]
end define

define class_head_rest
   [type_name] [base_clause?]
   | [nested_name_specifier] [type_name] [base_clause?]
end define

define class_key
   'class
   | 'struct
   | 'union
end define

define member_specification
   [member_declaration]
   | [access_specifier] ': [comment*]
   % this was included to allow for preprocessing directives inside
   % class definitions
   | [if_line] [comment*]
end define

define member_declaration
   [simple_member_declaration]
   | [function_definition] ['; ?] [comment*]
   | [':: ?] [nested_name_specifier]
        ['template ?] [unqualified_id] '; [comment*]
   | [using_declaration]
   | [template_declaration]
end define

define simple_member_declaration
   [member_declarator,] '; [comment*]
   | [decl_specifier] [simple_member_declaration]
end define

define member_declarator
   [wide_declarator]
   | [wide_declarator] [pure_specifier]
   | [wide_declarator] [constant_initializer]
   | [id ?] ': [conditional_expression]
end define

define pure_specifier
   '= 0
end define

define constant_initializer
```

```
   '= [conditional_expression]
end define

%=============================================================================
% A.9 Derived classes                                    [gram.class.derived]
%=============================================================================

define base_clause
   ': [comment*] [base_specifier,+]
end define

define base_specifier
   [':: ?] [nested_name_specifier?] [type_name]
   | 'virtual [access_specifier?] [':: ?] [nested_name_specifier?] [type_name]
   | [access_specifier] ['virtual ?] [':: ?]
                                        [nested_name_specifier?] [type_name]
end define

define access_specifier
   'private
   | 'protected
   | 'public
end define

%=============================================================================
% A.10 Special member functions                                [gram.special]
%=============================================================================

define conversion_function_id
   'operator  [conversion_type_id]
end define

define conversion_type_id
   [type_specifier+] [conversion_declarator?]
end define

define conversion_declarator
   [ptr_operator] [conversion_declarator?]
end define

define ctor_initializer
   ': [mem_initializer,+]
end define

define mem_initializer
   [comment*] [mem_initializer_id] '( [expression?] ')
end define

define mem_initializer_id
   [':: ?] [nested_name_specifier?] [type_name]
   | [id]
end define

%=============================================================================
% A.11 Overloading                                               [gram.over]
%=============================================================================

define operator_function_id
   'operator [operator]
end define

define operator
   'new
   | 'delete
   | 'new '[ ']
   | 'delete '[ ']
   | '+
   | '-
   | '*
   | '/
   | '%
   | '^
   | '&
   | '|
   | '~
   | '!
```

```
   | '=
   | '<
   | '>
   | '+=
   | '-=
   | '*=
   | '/=
   | '%=
   | '^=
   | '&=
   | '|=
   | '<<
   | '>>
   | '>>=
   | '<<=
   | '==
   | '!=
   | '<=
   | '>=
   | '&&
   | '||
   | '++
   | '--
   | ',
   | '->*
   | '->
   | '( ')
   | '[ ']
end define

%=============================================================================
% A.12 Templates                                                  [gram.temp]
%=============================================================================

define template_declaration
   ['export ?] 'template '< [template_parameter,+] '> [comment*] [declaration]
end define

define template_parameter
   [type_parameter]
   | [parameter_declaration]
end define

define type_parameter
   [comment*] 'class [id?] [comment*]
   | [comment*] 'class [id?] '= [type_id] [comment*]
   | [comment*] 'typename [id?] [comment*]
   | [comment*] 'typename [id?] '= [type_id] [comment*]
   | [comment*] 'template '< [template_parameter,+] '> 'class [id?] [comment*]
   | [comment*] 'template '< [template_parameter,+] '> 'class [id?]
                                                '= [id_expression] [comment*]
end define

define template_argument
   [assignment_expression]
   | [type_id]
   | [id_expression]
end define

define explicit_instantiation
   'template [comment*] [declaration]
end define

define explicit_specialization
   'template '< '> [comment*] [declaration]
end define

%=============================================================================
% A.13 Exception handling                                        [gram.except]
%=============================================================================

define try_block
   'try [comment*] [compound_statement] [handler+]
end define

define function_try_block
```

```
   'try [ctor_initializer?] [comment*] [function_body] [handler+]
end define

define handler
   'catch '( [exception_declaration] ') [comment*] [compound_statement]
end define

define exception_declaration
   [type_specifier+] [declarator]
   | [type_specifier+] [abstract_declarator]
   | [type_specifier+]
   | '...
end define

define throw_expression
   'throw [assignment_expression?]
end define

define exception_specification
   'throw '( [type_id,] ')
end define

%============================================================================
% A.14 Preprocessing directives                                    [gram.cpp]
%============================================================================

define pp_directive
   [if_line] [comment*]
   | [control_line] [comment*]
end define

define if_line
   '# 'if [conditional_expression]     '~>
   | '# 'ifdef [id]                  '~>
   | '# 'ifndef [id]                 '~>
   | '# 'elif [conditional_expression] '~>
   | '# 'else                        '~>
   | '# 'endif                       '~>
end define

define control_line
   '# 'include [preprocessing_token+]              '~>
   | '# 'includh [preprocessing_token+]            '~>
   | '# 'define [id] [replacement_list]            '~>
   | '# 'dmacro [id] '( [id,] ') [replacement_list]'~>
   | '# 'undef [id]                                '~>
   | '# 'line [preprocessing_token+]               '~>
   | '# 'error [preprocessing_token+]              '~>
   | '# 'pragma [preprocessing_token+]             '~>
   | '#                                            '~>
end define

define replacement_list
   [preprocessing_token*]
end define

%============================================================================

% for TXL

define program
   [translation_unit]
end define

% end of cpp.grm
```

cpp_filesummary.xslt

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE xsl:transform [<!ENTITY nbsp " ">]>
<!--

   PUC-Rio - Departamento de Informática

   Arquivo:    cpp_filesummary.xslt
   Projeto:    Lux
   Conteúdo:   Gerador de sumários para fontes C++
   Autor:      Marcelo Jaccoud Amaral
   Data:       2/5/2001

   Descrição:
      Esta transformação gera um documento XHTML 1.0 que
      descreve resumidamente as classes C++ (construções do
      tipo class, struct ou union) contidas no arquivo de
      entrada, que deve ser um documento que siga o DTD Lux
      para C++ (arquico cpp.dtd).
      Caso a classe possua uma descrição no arquivo-irmão RDF
      ([nome-do-fonte].dtd), esta descrição é incluída.
      O arquivo XHTML gerado usa a folha de estilos
      cpp_filesummary.css para uma melhor visualização.

-->

<xsl:transform version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:dc="http://purl.org/dc/elements/1.1"
   xmlns:lux="http://www.inf.puc-rio.br/lux/rdf/0.1"
   >

<xsl:output
   method="xml"
   version="1.0"
   encoding="iso-8859-1"
   indent="yes"
   doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"
   doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
   />

<!-- nome do arquivo com o código-fonte original -->
<xsl:variable name="source" select="/translation-unit/@source"/>

<!-- nome do arquivo com o hiperfonte correspondente (a entrada) -->
<xsl:variable name="xmlname" select="concat($source, '.xml')"/>

<!-- nome do arquivo RDF associado -->
<xsl:variable name="rdfname" select="concat($source, '.rdf')"/>

<!-- armazena o conteúdo do arquivo RDF -->
<xsl:variable name="rdftree" select="document($rdfname)"/>

<!--
   Gabarito para o elemento-raiz.
   Insere os elementos XHTML principais, incluindo o título e
   o link para a folha de estilos CSS2, e depois processa os
   elementos class-specifier, que são as definições de classe.
-->
<xsl:template match="translation-unit">
   <html xmlns="http://www.w3.org/1999/xhtml">
      <head>
         <title>Sumário de classes</title>
         <link rel="stylesheet" href="cpp_filesummary.css" />
      </head>
      <body>
         <h1>Classes em «<xsl:value-of select="$source"/>»</h1>
         <xsl:apply-templates select=".//class-specifier"/>
      </body>
   </html>
</xsl:template>
```

```
<!--
   Gabarito para uma declaração de classe.
   Insere o nome da classe, seu espaço nominal e número de
   membros, e sua eventual descrição.
-->
<xsl:template match="class-specifier">
   <xsl:apply-templates select="class-head"/>
   <xsl:call-template name="get-namespace"/>
   <p>
      <xsl:text>Número de membros: </xsl:text>
      <xsl:value-of select="count(member-declaration)"/>
   </p>
   <xsl:call-template name="get-description"/>
</xsl:template>

<!-- Cabeçalho da classe. Formata seu nome e classes-bases -->
<xsl:template match="class-head">
   <h2>
      <xsl:apply-templates select="*[not(name()='base-specifier')]"/>
   </h2>
   <xsl:apply-templates select="base-specifier"/>
</xsl:template>


<!--
   Gabarito para elementos vazios. Gera um texto com o nome do elemento.
-->
<xsl:template match="class | struct | union | template |
                     virtual | private | protected | public">
   <i><xsl:value-of select="name()"/> </i>
</xsl:template>


<!--
   Os seguintes gabaritos permitem gerar o nome correto dos tipos.
-->
<xsl:template match="nested-namespace-specifier">
   <xsl:for-each select="*">
      <xsl:apply-templates/>
      <xsl:text>::</xsl:text>
   </xsl:for-each>
</xsl:template>

<xsl:template match="template-spec">
   <xsl:text>&lt;...&gt;</xsl:text>
</xsl:template>

<xsl:template match="base-specifier">
   <p>derivação: <xsl:apply-templates/></p>
</xsl:template>

<xsl:template match="global">
   <xsl:text>::</xsl:text>
</xsl:template>

<xsl:template match="nm">
   <xsl:value-of select="."/>
</xsl:template>


<!-- ****** Gabaritos com nome ******* -->

<!--
   Recupera o namespace no qual a classe foi definida.
   Isto é feito subindo na árvore em direção à raiz e selecionando o
   primeiro elemento namespace-definition que aparecer.
   (A rigor, isso deve ser feito recursivamente, pois as declarações de
   espaços nominais são aninháveis. Isso não costuma ocorrer na prática,
   entretanto, e este gabarito é só um exemplo sem maiores pretensões
   para ilustrar como recuperar informação de outra parte do documento.)
-->
<xsl:template name="get-namespace">
   <xsl:variable name="ns-decl" select="ancestor::namespace-definition"/>
   <xsl:if test="$ns-decl">
      <p><xsl:text>namespace: </xsl:text>
```

```
      <xsl:variable name="ns-name" select="$ns-decl/nm"/>
      <xsl:choose>
         <xsl:when test="$ns-name">
            <xsl:value-of select="$ns-name"/>
         </xsl:when>
         <xsl:otherwise>(anônimo)</xsl:otherwise>
      </xsl:choose>
      </p>
   </xsl:if>
</xsl:template>

<!--
   Recupera as descrições RDF associadas ao elemento passado como
   parâmetro (o elemento presuntivo é o corrente). A busca é feita procurando
   pelo ID do elemento na árvore RDF, que está armazenada na variável $rdftree.
   Este gabarito ilustra como recuperar informações de outros documentos XML.
-->
<xsl:template name="get-description">
   <xsl:param name="index" select="@id"/>
   <xsl:variable  name="rdf-entry"
      select="$rdftree//rdf:Description[@about=concat($xmlname,'#',$index)]"/>
   <xsl:if test="$rdf-entry">
      <p class="description">
         <xsl:value-of select="$rdf-entry/lux:description"/>
      </p>
   </xsl:if>
</xsl:template>

</xsl:transform>
<!-- fim de cpp_filesummary.xslt -->
```

cpp_filesummary.css

```
/*
   PUC-Rio - Departamento de Informática

   Arquivo:    cpp_filesummary.css
   Conteúdo:   Uma folha de estilos simples para o resumo de classes
               gerado pela transformação cpp_filesummary.xslt.
   Autor:      Marcelo Jaccoud Amaral
   Data:       02/05/2001

*/



html {
   font-size: 12pt;
   font-family: Times, serif;
}

h1 {
   margin: 0pt;
   padding: 2pt;
   font-size: 16pt;
   background-color: teal;
   color: white;
   text-align: center;
}

h2 {
   margin-top: 6pt;
   margin-bottom: 6pt;
   font-size: 14pt;
}

h4 {
   margin-top: 4pt;
   margin-bottom: 4pt;
   font-size: 12pt;
}

p {
   padding: 4 pt;
   margin-left: 3%;
   margin-top: 0pt;
   margin-bottom: 0pt;
   }

/*
   O estilo para a descrição é semelhante ao usado por rdf.css.
   Note que é fundamental que se usem fontes de espaçamento fixo, pois
   estes comentários foram importados de arquivos-texto. Caso se façam
   edições nas descrições e estas venham a ser textos marcados, deve-se
   usar um estilo diferente, com fontes de espaçamento proporcional.
   A folha de transformação XSLT pode selecionar o estilo correto baseado
   no atributo xml:space, que vale sempre 'preserve' para os comentários
   importados.
*/

p.description {
   font-family: "Andale Mono", monospaced;
   font-size: 9pt;
   background-color: #FFFFCC;
   display: block;
   white-space: pre;
   border: 1.5pt silver solid;
   margin-top: 6pt;
   padding-bottom: 4pt;
  }

p.description:before {
   font-family: Arial, sans-serif;
   font-size: 10pt;
   content: "Descrição:";
```

```
   display: block;
   padding-left: 2pt;
   font-weight: bold;
   background-color: silver;
   border: 1pt silver solid;
}

/* fim de cpp_filesummary.css */
```

GLOSSÁRIO

alfabeto Um conjunto de símbolos que geralmente representa sons numa língua, como o alfabeto latino usado pelos idiomas ocidentais. Contudo, não há uma relação simples entre os símbolos (letras) e sons. O padrão Unicode define explicitamente quais caracteres são consideradas alfabéticos.

âncora Qualquer uma das extremidades de um *link* num documento.

API *(Application Programming Interface)* Uma interface padronizada para acesso aos componentes internos de uma biblioteca ou programa. É normalmente constituída de um conjunto de estruturas de dados e rotinas de acesso (em linguagens orientadas a objeto, um conjunto de classes).

aplicativo Um módulo de *software* em favor do qual o processador XML lê e escreve dados em XML. Por exemplo, uma aplicativo de banco de dados pode usar um processador XMl para ler e escrever dados no formato XML. Isto não deve ser confundido com a definição SGML de aplicação, que equivale à uma DTD determinada.

ASCII *(American Standard Code for Information Interchange)* Um conjunto de caracteres codificado de 7 *bits* representando 128 caracteres. Vinte e três desses são caracteres de controle usados no controle de comunicação e impressão. Somente três destes — o tabulador horizontal (TAB), o avanço de linha (LF) e o retorno de carro (CR) — são admitidos num documento XML.

atributo Termo predicativo que pode ter um valor e é associado a um elemento. Os

valores dum atributo são comumente indicados no marcador inicial ou no marcador de elemento vazio. Por exemplo, `<obj attr="val">` significa que o elemento é do tipo obj e possui um atributo de nome attr com valor "val". Os atributos não são ordenados e cada um só deve aparecer uma única vez. Na declaração da lista de atributos, cada um pode receber um valor presuntivo, ser declarado como fixo (não pode ter o valor alterado) ou requerido (deve ser especificado no marcador).

bem formado Um documento XML é dito bem formado se é consistente com a gramática da especificação XML, independentemente de uma DTD. Um documento bem formado é processável por qualquer *parser* XML.

caráter ou caractere Uma palavra ambígua, com muitas definições. Em computação, é melhor definido como uma quantidade que representa uma função abstrata associada com um caráter, independente de qualquer representação gráfica. Alternativamente, pense num caráter como uma unidade básica que representa um dado ou códigos usados para organizar e controlar dados. Para uma discussão detalhada, veja [Graham] e o padrão [Unicode].

CASE *Computer Aided Software Engineering*. Designação para uma classe de aplicativos destinados à automação dos processos de concepção, projeto e implementação de *software*.

classe de caracteres Um termo usado para distinguir diferentes subconjuntos de caracteres, segundo características definidas no padrão Unicode. As seguintes classes são definidas no padrão: caracteres básicos (que contém, entre outros, os caracteres alfabéticos latinos, sem sinais diacríticos), caracteres ideográficos, caracteres combinantes (que se combinam com os precedentes, na sua maioria sinais diacríticos); estas classes, unidas, formam a classe das

letras. Outras classes são os dígitos (caracteres que se referem a números) e extensores (um tipo de caráter combinante que afeta o significado do precedente, usualmente alterando seu senso de tamanho ou largura associada ao significado do caráter).

codificação de caracteres Um mapeamento entre um conjunto de caracteres e um conjunto de *bytes.* Um conjunto de caracteres pode suportar mais de um esquema de codificação. Por exemplo, o conjunto de caracteres Unicode pode ser codificado de forma a usar sempre dois octetos (UTF-16, uma codificação de 16 *bits*), de um a quatro octetos (UTF-8, uma codificação de 8 *bits*), ou de um a quatro octetos, fazendo uso apenas dos sete *bits* menos significantes em cada octeto (UTF-7, uma codificação de 7 *bits*).

conjunto [codificado] de caracteres Um conjunto de regras inequívocas que estabelece um conjunto de caracteres e um mapeamento entre os caracteres e uma seqüência de números inteiros. É comum dizer-se simplesmente "conjunto de caracteres".

conteúdo Os caracteres e a marcação contidas entre os marcadores inicial e final de um elemento.

documento conforme Um documento que respeita a gramática e as restrições da especificação XML.

DOM *(Document Object Model)* Uma API orientada a objeto para documentos XML e HTML, que consiste de classes com métodos para acesso e modificação da árvore do documento, seus elementos e atributos.

EBCDIC Um conjunto codificado de caracteres originado na década de 60 nos antigos *mainframes* da IBM.

EBNF *(Extended Backus-Naur Form)* Notação (uma gramática livre de contexto) usada para expressar as regras sintáticas de uma linguagem.

elemento radical ou elemento-raiz O elemento único que contém todos os demais e compõe o corpo do documento XML.

elemento. Unidade lógica de um documento XML, delimitado por marcadores. Os elementos são organizados numa árvore, a partir do elemento-raiz. Um elemento pode possuir pares atributo-valor e conteúdo. O nome do elemento é indicado nos seus marcadores, e indica o tipo do elemento.

enleio Processo de geração de código compilável a partir dum hiperfonte ou programa letrado.

entidade Fisicamente, um documento XML é composto por unidades chamadas de entidades, sendo o próprio documento a raiz ou entidade radical. A entidade radical pode fazer referência a outras, que lhe são incluídas, e estas podem fazer referências a outras, etc.

espaço nominal (ingl. *namespace*) Um espaço abstrato no qual um nome tem um significado definido. Mecanismos que definem espaços diferentes e permitem referenciar elementos homônimos em diferentes espaços são usados por algumas linguagens de programação (e.g. C++). XML define um mecanismo próprio que permite usar num mesmo documento elementos de diversas DTDs sem que haja conflito em decorrência de elementos homônimos.

fonte Uma coleção de glifos tendo o mesmo estilo básico, e.g., Courier Negrito Oblíquo.

gabarito (ingl. *template*) Em XSLT, um conjunto de elementos e instruções que pode

ser aplicado a um conjunto de elementos. Um gabarito pode ser invocado como efeito do resultado de um casamento de padrões sobre uma expressão seletora de elementos (XPath) ou diretamente, se possuir um nome. Um gabarito é aplicado sobre o contexto resultante da expressão seletora, que presuntivamente é igual ao contexto corrente.

glifo Um símbolo gráfico que provê aparência e forma para um caráter. Um glifo possui forma distinta, mas nenhum significado especial à parte de sua relação com um caráter. Segundo o padrão Unicode, *" a recognizable abstract graphic symbol which is independent of any specific design"*.

HTML *HyperText Markup Language.* Linguagem desenvolvida segundo o padrão SGML para criar documentos com hipertexto, e largamente utilizada na Internet.

HTTP *(HyperText Transfer Protocol)* Protocolo de comunicação projetado especialmente para distribuir documentos em hipertexto e dados relacionados a eles. o protocolo (versão 1.1) é definido em RFC 2068.

ISO Latin-1 O mesmo que ISO/IEC 8859-1.

marcação Texto que é inserido ao conteúdo de um documento com o intuito de prover informação sobre a estrutura lógica dos dados textuais.

marcador Cada um dos *tokens* que delimita um elemento XML. Também conhecidos como rótulos (ingl. *tag*). O marcador inicial de um elemento `x` tem a forma `<x>` e o marcador final `</x>`. Um elemento vazio (sem conteúdo) pode tanto ser escrito `<x></x>` como usando a abreviação `<x/>`. O marcador inicial pode conter atributos.

modelo de conteúdo Um conjunto de regras para o aninhamento de elementos e

texto num documento, especificadas nas declarações duma DTD.

navegador Um aplicativo para visualizar hipertexto, que permite seguir facilmente os vínculos entre os documentos, carregando-os sob demanda.

octeto Uma unidade de armazenamento de informação cujo comprimento é 8 *bits*. as especificações ISO/IEC dos conjuntos de caracteres usam o termo octeto ao invés de *byte*, pois alguns computadores definem *byte* com comprimentos diferentes.

postscript Uma linguagem de programação voltada para a diagramação e leiaute de páginas, usada principalmente em dispositivos de impressão. Muitos sistemas usam postscript como formato padrão de saída.

processador Um módulo de programa, usualmente construído como uma biblioteca visando o reúso, que lê e/ou escreve arquivos num formato específico. Os mais comuns são os analisadores sintáticos (*parsers*). Os processadores se comunicam com os demais módulos de um aplicativo por meio de uma interface padrão, isolando-os dos detalhes envolvidos na descodificação dos documentos.

referência característica Uma referência para um caráter específico, usando seu índice decimal ou hexadecimal no conjunto de caracteres universal definido no padrão Unicode. Por exemplo, as referências características `&#233;` (decimal) e `&#xE9;` (hexadecimal) se referem ambas ao caráter "e com acento agudo" (é). A referência característica pode ser encarada como um mecanismo de escape para caracteres Unicode.

SGML *Standard Generalized Markup Language.* Padrão ISO para a criação de linguagens de marcação.

***software* de autoria** Qualquer sistema que pode ser usado para criar e editar docu-

mentos XML. Pode variar desde um simples editor de textos a um complexo editor especializado em XML, que imprime corretude gramatical e pode validar um documento frente à sua DTD.

tecedura Processo de gerar documentação detalhada a partir dum hiperfonte ou programa letrado.

texto XML define texto como uma seqüência de caracteres, que podem representar marcação ou dados textuais.

tipo Um *design* para um conjunto de glifos que compartilham um mesmo estilo. Cada instância de um tipo, com tamanho e estilo específicos, é chamado fonte. Uma coleção de fontes com tipos que seguem o mesmo *design* (com variações no tamanho, inclinação, natureza do itálico, etc.) são chamados membros de uma família de tipos. Por exemplo, a fonte "Helvética Negrito 10 pt" é um membro da família de tipos Helvética.

token Uma seqüência de caracteres logicamente coesiva, como um identificador, uma palavra-chave (como `if` ou `while`) , um caráter de pontuação ou um operador composto de um ou mais caracteres (como `:=`). Os *tokens* são os primeiros elementos a serem identificados por um analisador léxico.

UCS *(Universal Character Set)* Neste trabalho, refere-se ao conjunto universal unificado de caracteres Unicode – ISO/IEC 10646.

UCS *Universal Character Set*. Neste trabalho, refere-se ao conjunto de caracteres unificado Unicode / ISO/IEC 10646. Tanto XML como HTL 4.0 estabelecem este conjunto como normativo.

UCS-2 Uma codificação particular do UCS, onde cada caráter é codificado em dois

bytes.

UML *Unified Modeling Language*. Uma linguagem predominantemente visual voltada para a modelagem conceptual e arquitetural e o projeto de sistemas, em particular os orientados a objeto. Vem sendo adotada por várias processos de desenvolvimento e ferramentas CASE.

URI *Universal Resource Identifier.* Termo genérico para uma cadeia de caracteres codificada que identifica um recurso, tipicamente na Internet. Há três tipos de exemplos práticos de URIs: URLs, URLs parciais (também chamadas de URLs relativas) e URNs *(Uniform Resource Names)*.

URL *Uniform Resource Locator.* Esquema usado para endereçar recursos da Internet na World Wide Web. O URL especifica o protocolo, nome de domínio ou endereço IP, número da porta, rota e detalhes do recurso necessários para atingi-lo de uma determinada máquina. URLs parciais são um esquema que permite usar outro recurso como ponto de partida, especificando uma rota relativa de acesso.

WYSIWYG *("What You See Is What You Get")* Acrônimo inglês usado para designar interfaces de edição onde o usuário vê e trabalha no dispositivo de edição (em geral um monitor de vídeo) com uma representação fiel de como o documento será produzido no dispositivo final (uma impressora ou fotocompositora).

XML *Extensible Markup Language*. Um padrão criado e mantido pelo W3C que permite criar linguagens de marcação. Define um conjunto genérico de regras que torna a sintaxe e a codificação dos documentos uniforme, dando completa portabilidade ao texto.

XSL *XML Stylesheet Language.* Uma linguagem criada para formatar documentos

XML. É definida por dois padrões distintos: XSLT, que trata de transformações gerais da estrutura do documento; e XSL-FO, que define objetos para formatação física.

XSL-FO *XSL Formating Objects.* Parte da linguagem XSL que trata da formatação física do texto, tanto para impressão (tipos, margens, etc.) como outros meios (e.g. formatação aural).

XSLT *XSL Transformations.* Parte da linguagem XSL que trata de transformações na estrutura de marcadores de um documento XML. Uma transformação XSLT pode gerar tanto outro documento XML como documentos HTML ou texto não marcado.