Carlos Roberto Serra Pinto Cassino

Uma Ferramenta para Programação Literária Modular

Dissertação de Mestrado

Departamento de Informática

Rio de Janeiro, 12 de agosto de 1996

Carlos Roberto Serra Pinto Cassino

Uma Ferramenta para Programação Literária Modular

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

Orientador: Roberto Ierusalimschy

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro

Rio de Janeiro, 12 de agosto de 1996

Para meus pais,
Fernando & Neide
e para Marcela.

Agradecimentos

Ao Roberto, orientador e amigo que soube guiar não só esse trabalho mas também meu desenvolvimento profissional ao longo desses dois anos.
Ao André Costa, ao André Clinio, ao Renato Borges e ao Renato Cerqueira, que foram amigos sinceros e me deram toda a ajuda e o apoio de que precisei.
Ao TeCGraf que garantiu um excelente ambiente para a execução desse trabalho.
Ao CNPq pelo auxílio financeiro ao longo do curso.

Resumo

Esse trabalho apresenta uma ferramenta de suporte à programação literária. O desenvolvimento dessa ferramenta foi precedido por um estudo dos serviços básicos normalmente requeridos por programas dessa classe. Após esse estudo, foi desenvolvido um framework para oferecer esses serviços.

O framework foi desenvolvido em C++ padrão, de modo a garantir sua portabilidade. Sua eficiência foi obtida através de um núcleo otimizado para realizar as operações a nível de caracter, enquanto uma interface de programação (API) bem definida garante sua flexibilidade. Sobre esse framework de classes C++ foi criada uma camada de acesso via uma linguagem de configuração. Essa camada permite que, através de pequenos programas escritos na linguagem de configuração, um usuário possa adaptar a ferramenta para diferentes estilos de programação literária, como por exemplo documentos modulares ou navegáveis via Internet.

Abstract

This thesis presents a support tool for literate programming. Its development was preceded by a research of the basic facilites usually required for programs of this class. After that, a framework has been developed to offer those facilites.

The framework was developed in standard C++, in such a way that would guarantee its portability. Its efficiency is due to an optimzed kernel which performs all character-level operations, and a well defined application program interface (API) assures its flexibility. On top of this framework an access layer has been created using a configuration language. This layer allows that, through small programs written in the configuration language, a user can adapt the tool for different literate programming styles, like modular or WWW-enabled documentation.

Índice

Capítulo 1

Introdução

1.1 Programação Literária

O termo "Literate Programming" foi inventado por Donald Knuth [Knu84] para designar a técnica que, segundo ele, seria ideal para a construção de um programa bem documentado. Sua idéia atribui à documentação uma importância tão grande quanto a do próprio código: "Vamos mudar nossa atitude tradicional com relação à construção de programas. Ao invés de imaginar que nossa tarefa principal é instruir a um *computador* o que fazer, vamos nos concentrar em explicar a *seres humanos* o que nós queremos que o computador faça." [Knu84], pág. 97. Com a mudança de paradigma, programas deixam de ser grandes porções de código entremeadas com comentários e passam a ser grandes porções de documentação entremeadas com código.

Esse método de construção de programas sugere que um programador deve lidar com pelo menos duas linguagens diferentes durante o processo de criação: uma linguagem para programar e outra para documentar. O programador deve criar seu programa como uma narrativa dos fatos que determinam sua estrutura. A narrativa segue, definindo, por exemplo, as estruturas de dados que, por sua vez, são escritas na linguagem de programação; de maneira análoga, todo o programa vai surgindo como consequência da documentação que o especifica. Mais ainda, a programação literária não amarra o programador a um determinado estilo de programação do tipo "top-down" ou "bottom-up": a apresentação é feita segundo a ordem mais natural para o desenvolvedor, tornando mais clara a apresentação do trabalho.

Com essa forma de documentação, espera-se que o número de erros de programação diminua sensivelmente, visto que o programador estará sempre explicando as alternativas adotadas: alternativas ruins simplesmente não terão explicações convincentes e serão automaticamente repensadas. Analogamente, em um nível menor de abstração, pequenos erros de codificação também serão diminuídos frente a uma explicação prévia da funcionalidade do código escrito. Outro benefício é permitir que o código seja lido sem grande esforço por novos programadores que venham a trabalhar, seja com desenvolvimento seja com manutenção, no projeto — meta importante de qualquer documentação.

Para poder usar o método proposto, Knuth implementou um sistema chamado WEB, composto por dois programas: `weave` e `tangle`. A idéia era permitir que o usuário criasse o programa segundo o método exposto e pudesse, sem esforços adicionais, extrair daí tanto o código quanto a documentação final. Neste sistema, Knuth adotou duas linguagens fixas para desenvolvimento: TEX para escrever e formatar a documentação, e Pascal como linguagem de programação. `weave` é o programa encarregado de transformar o texto escrito pelo programador em um arquivo no formato TEX. Esse arquivo TEX contém todas as formatações necessárias para que o texto seja compilado e impresso, gerando a documentação do programa literário. Já o programa `tangle` faz a extração do código, a partir do mesmo fonte escrito pelo programador. O código extraído é enviado para um arquivo que, compilado, gerará o programa executável.

Após o sistema WEB, diversos trabalhos foram divulgados na lituratura sobre esta técnica mas, como observou Christopher Van Wyk [Van90], a maior parte das pessoas que a adotavam criava sua própria ferramenta de apoio. Esse fato foi interpretado por Van Wyk como um indicativo de que essa técnica não estava realmente difundida e aceita pela comunidade. Posteriormente à publicação de seu artigo, novas ferramentas foram criadas por diferentes grupos, reforçando sua linha de raciocínio.

As razões para esta proliferação de ferramentas são diversas. A principal delas diz respeito às linguagens adotadas. WEB só suporta programas em Pascal; para programas escritos em C, foi desenvolvido CWEB [KL93]. Várias outras ferramentas foram desenvolvidas para várias outras linguagens. Problemas com portabilidade e eficiência também levaram vários grupos a desenvolver ferramentas específicas para suas necessidades. A falta de modularidade da maioria desses sistemas também dificulta sua adaptação para outros fins. Além disso, as ferramentas de programação literária não oferecem bom suporte para modularização.

1.2 Apresentação do Trabalho

De modo geral, uma ferramenta de apoio à programação literária é dividida em dois programas, um para extrair o código e um para gerar a documentação. O programa de extração de código percorre o arquivo fonte escrito pelo usuário, identificando os trechos de código e gerando, ao término, um arquivo contendo o código a ser compilado para se obter o programa desejado. Já o programa que gera a documentação processa o mesmo fonte de uma maneira diferente, adicionando informações e formatações necessárias a transformá-lo em um fonte a ser processado por um formatador de textos para, então, se obter a documentação final.

Para permitir o processamento pela ferramenta, o arquivo fonte deve conter, além do código e da documentação, comandos ou marcas definidas pela própria ferramenta cujo objetivo é o de separar e, possivelmente, identificar os diversos trechos de documentação e de código. É com base nesses comandos, ou marcas, que a ferramenta separa as partes, armazenando-as, se necessário, e as processa, gerando o resultado final. Esse tipo básico de operação realizada por uma ferramenta sugere uma forma de macro-processamento do fonte, associado a um processamento algorítmico das partes identificadas, a fim de gerar o resultado final.

Este trabalho apresenta a ferramenta **nome**, de apoio à programação literária. Entretanto, diferentemente de propostas anteriores, o desenvolvimento desta ferramenta foi precedido pela construção de um framework que disponibiliza os serviços básicos realizados por esse tipo de ferramenta. Com essa atitude, estamos assumindo a necessidade de criação de novas ferramentas. Ao invés de apenas criarmos mais uma ferramenta sobre a qual os programadores devam se adaptar, estamos criando uma *base* sobre a qual tais ferramentas possam ser mais facilmente construídas.

Para que o framework possa ser usado como base de novos desenvolvimentos, os serviços básicos de macro-processamento foram disponibilizados a uma linguagem de configuração/extensão. Esse passo representa uma abstração que facilita a criação de uma ferramenta, a qual passa a ser um pequeno programa escrito na linguagem de extensão. Prover os serviços básicos de macro-processamento por intermédio de uma linguagem de extensão traz ao framework uma grande flexibilidade, disponibilizando estruturas de dados e processamento algorítmico, recursos que tipicamente não estão disponíveis nos macro-processadores comuns. Essa flexibilidade é fundamental para permitir a construção de ferramentas com características bastante variadas. Além disso, a arquitetura do framework garante eficiência e portabilidade às ferramentas desenvolvidas.

A ferramenta **nome** exemplifica as características do framework. A flexibilidade foi explorada configurando-se a ferramenta para aceitar programas literários desenvolvidos para o sistema NOWEB [Ram92], ao mesmo tempo ampliando seu funcionamento com facilidades para a construção de documentos de forma modular. Eficiência e portabilidade foram obtidas ao final, tornando a ferramenta mais portável e dotando-a de desempenho superior ao da ferramenta usada como molde.

A estrutura deste documento reflete a do próprio estudo. O capítulo 2 apresenta um estudo de trabalhos correlatos, incluíndo análises de ferramentas existentes. A arquitetura do framework é discutida no capítulo 3, sendo o capítulo 4 destinado à apresentação de **nome**. O último capítulo é dedicado a uma avaliação do trabalho.

Capítulo 2

Trabalhos Correlatos

Na seção 2.1 são analisados alguns estudos sobre métodos de documentação de software. Foram selecionados: "Literate Programming" de Donald Knuth [Knu84] que, conforme citado na introdução, introduziu o conceito de programação literária. "Literate Programming on a Team Project" de Norman Ramsey e Carla Marceau [RM91] que fizeram uso desta metodologia em um projeto de software não trivial (em torno de 33.000 linhas de código). "Literate Smalltalk Programming using Hypertext" de Kasper Osterbye [Ost93] que apresenta um estudo sobre como hipertextos podem suportar a programação literária. "Integrating a Hypertext Interface into a Syntax-Directed Programming Environment" de Michael Bell e Mark Rivers [BR93] apresenta a integração de um sistema de hipertexto com um de programação, de forma a facilitar o acesso ao código e à documentação do projeto. Para terminar, "Uma Ferramenta para Geração de Documentação de Sistemas de Software" de Christiano Braga [dOB95], que propõe uma arquitetura híbrida para a geração de documentação hipertextual de programas.

Já na seção 2.2 são analisadas algumas ferramentas para suporte à programação literária. Foram selecionadas cinco ferramentas: CWEB, FWEB, CLiP, FunnelWeb e NOWEB. Para cada uma é feita uma análise, descrevendo suas principais características e tentando levantar, também, seus pontos fracos.

Por fim, a seção 2.3 apresenta alguns macro-processadores. Conforme será visto no capítulo 3, esse tipo de ferramenta possui alguma analogia com o que será desenvolvido neste trabalho.

2.1 Trabalhos Analisados

2.1.1 Literate Programming

Esse trabalho [Knu84] introduz o termo "programação literária" como uma metodologia para a criação de programas bem documentados. Uma mudança de conceito é necessária: programas devem ser vistos como trabalhos de literatura. Essa mudança significa que programas não serão mais vistos como 'instruções a um computador do que fazer' mas como 'explicações a um ser humano do que desejamos que o computador faça'. Dessa forma, passamos a ter o programa como um resultado de uma documentação que o especifica.

Para implementar a idéia, Knuth desenvolveu uma ferramenta capaz de processar o programa escrito, gerando o código a ser compilado e a documentação a ser impressa. É importante, então, que os arquivos a serem processados sigam um padrão estabelecido,

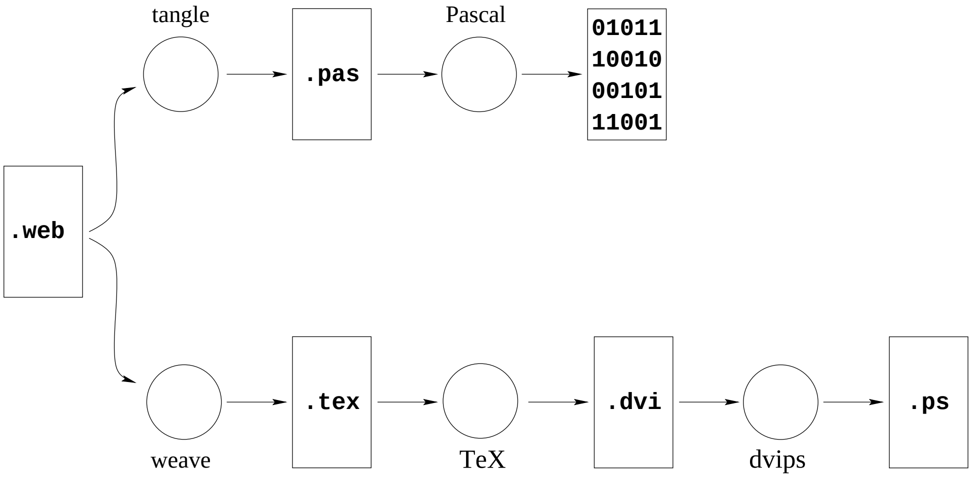

Figura 2.1: Esquema de uso da ferramenta WEB.

sendo estruturados por meio de uma linguagem da própria ferramenta. Essa linguagem da ferramenta dá meios ao programador de separar, dentro do documento único, o que é código do que é documentação. A ferramenta introduz, então, um passo adicional do processo normal de codificação/compilação/teste, como pode ser visto na figura 2.1.

O documento criado pelo usuário deve seguir, então, um padrão estabelecido, dividindo-se em seções que representam cada uma das partes nas quais o programa pode ser conceitualmente dividido. A abordagem utilizada é a de que um programa complexo pode ser visto como um conjunto de partes, de complexidade reduzida, cuja integração implementa a funcionalidade desejada. Dessa forma, o entendimento do todo será alcançado a partir do entendimento das partes e dos relacionamentos entre tais partes. Segundo Knuth, uma leitura do documento, começando pela primeira seção e seguindo seqüencialmente até a última, trará um entendimento do programa da forma mais agradável possível, bastando apenas que, para isso, o programador tenha escolhido a forma mais natural para a apresentação.

Cada uma das seções das quais se compõe o documento pode ser sub-dividida em três partes componentes. A primeira parte, de descrições, deve ser uma explicação do material que se segue na seção, constituíndo-se apenas de texto, possivelmente formatado por comandos do $\TeX$. A segunda parte contém macro-definições no mesmo sentido das macro-definições usadas em programas C — uma característica ausente em Pascal mas que é disponibilizada pela ferramenta de Knuth. Essa parte é, então, composta de um misto de comandos da ferramenta e código Pascal. Por fim segue-se uma parte contendo um trecho de código que implementa a funcionalidade descrita. Essa última parte é composta não apenas de código Pascal mas também de comandos da ferramenta: são esses comandos que permitirão ao programador usar as macros definidas, bem como fazer referência às demais seções. Cada uma dessas partes pode se apresentar vazia, facilitando, assim, a apresentação pelo programador. Dessa forma, uma primeira seção, por exemplo, pode não conter nem macros e nem código, sendo uma explicação em mais alto nível do que se segue no resto do documento. Já uma seção destinada a implementar um pequeno e código que se explica por sí só, pode não conter descrições nem macro-definições.

Cada seção possui um nome pelo qual ela pode ser referenciada. Uma referência a uma seção, feita de dentro da parte de código de uma seção qualquer, resulta na expansão de seu código, no ponto onde foi feita a referência. Deve-se notar que a funcionalidade se assemelha à de macro-definições. A diferença fica por conta do nível de abstração, uma vez que seções são unidades lógicas com uma funcionalidade bem definida que, embora simples, necessitam de uma descrição ou alguma avaliação. Já macros são diminutos trechos de código que se explicam por sí só, usadas apenas para simplificar a apresentação de trechos de código mais complexos. Outro ponto importante é que as macros disponibilizadas pela ferramenta implementam um conceito comum em C mas inexistente em Pascal, disponibilizando, inclusive, a capacidade de efetuar cálculos de expressões constantes, algo que os compiladores Pascal não fazem.

A extração do código, executada pelo programa `tangle`, é feita a partir das seções que não foram nomeadas, sendo o código dessas seções expandidos diretamente para o arquivo de saída. Sobre o código gerado são buscadas todas as referências a outras seções, sendo o código correspondente expandido no local da referência. Em ambos os passos todas os usos das macros definidas pelo usuário são substituídos por seus valores.

`weave` se encarrega de transformar o texto escrito pelo programador em um arquivo no formato TEX. Neste passo, a formatação introduzida pelo autor é mantida e novas formatações são feitas, incluíndo a formatação do código: palavras reservadas em negrito, identificadores em itálico, símbolos matemáticos nas expressões etc. As seções são numeradas, sendo os números anexados a seus nomes para facilitar as referências e a busca pelo usuário. Todas as informações necessárias sobre definições e usos são escritas ao término de cada seção. São adicionadas formatações no início das seções principais (tal informação é dada pelo usuário através de comandos da ferramenta) para dar-lhes o destaque necessário. Também são criados: um índice que indica as seções nas quais aparecem cada identificador Pascal usado pelo programador; uma lista, em ordem alfabética, contendo todas as seções; e uma tabela de conteúdo mostrando o nome e a página das seções principais.

2.1.2 Literate Programming on a Team Project

Esse trabalho [RM91] é muito interessante pela crítica feita à técnica de programação literária. Tal crítica é resultado de um projeto, realizado durante três anos por uma equipe de sete programadores, de um sistema com 33.000 linhas de fonte — 13.000 das quais são documentação. A ferramenta utilizada pelo grupo de Norman Ramsey foi CWEB. Esta ferramenta, que será analisada na seção 2.2.1, é uma versão de WEB criada para permitir o uso de C como linguagem de programação.

Um ponto importante deste trabalho, como salientado logo em sua introdução, é o fato deste ser um projeto real, não um pequeno programa feito exclusivamente para publicação ou para fins didáticos, mas um programa feito com um objetivo prático. Além disso, o porte do programa exigiu um trabalho cooperativo — tópico não abordado nos demais artigos sobre o tema.

O programa desenvolvido foi um editor de auxílio ao desenvolvimento de programas formalmente verificados em Ada. Tendo em vista uma "forte tendência" a erros de programação no sistema (dada a complexidade das operações efetuadas), Ramsey optou pela programação literária. A idéia era evitar erros com uma implementação fortemente conectada à documentação e à especificação das funções. Além disso, o uso do modo matemático de TEX facilita muito a tarefa de escrever tais especificações.

Dentro deste cenário — um programa em pleno processo de desenvolvimento — a visão mais correta do programa, segundo Ramsey, é "não um livro, mas um manual de reparo de carros". A estrutura dada ao programa foi bastante parecida com a estrutura do manual de Ada. Desta forma, o texto do programa serviu como referência para os programadores e como material de apresentação para os novos programadores admitidos no projeto.

Segundo Ramsey, a ferramenta CWEB apresentou vários problemas:

- A inviabilidade da inclusão de diagramas e figuras. Essa característica dificulta (ou não facilita) a explicação de certas partes do programa.

- A impressão formatada do código não respeita as identações e as quebras de linha feitas pelos programadores. Após diversas alterações feitas para mudar a saída do `weave`, a conclusão final foi a de que se estaria melhor sem a impressão formatada.

- A integração entre CWEB e TEXé muito pequena. Com isso, não é possível extrair um trecho do documento gerado por `weave` e incluí-lo diretamente em um outro documento. Para fazer uma integração deste tipo, são necessários ajustes manuais nas macros — um trabalho adequado apenas para especialistas em TEX.

Um outro ponto é que um arquivo escrito no formato TEX não é de fácil leitura em virtude da presença de comandos do formatador no meio do texto. Segundo esse modo de operar, o documento escrito é bastante diferente da versão compilada e impressa. Isso não chega a trazer problemas quando se escreve um texto em que só a versão impressa será lida. Porém, o mesmo não se pode afirmar de um programa que deve ser consultado e editado diariamente por programadores.

A tabela de conteúdo gerada por `weave` também se mostrou deficiente por ser formada basicamente por uma lista de seções. Essa estrutura é muito plana para descrever programas não triviais e deveria ser estentida. Uma possível solução seria o uso da estrutura implementada pelo LATEX que inclui capítulos, seções e dois níveis de subseções mas, infelizmente, LATEX não pode ser usado como formatador em CWEB.

Além desses problemas específicos com a ferramenta, a equipe de Ramsey teve dificuldades com as técnicas a serem usadas para o desenvolvimento, visto que poucas são as existentes na literatura. Com dificuldades para criar um método preciso, a equipe procurou incluir as especificações do projeto ao longo da documentação do código. Usando essa política foi possível desenvolver o projeto por três anos com uma equipe variável sem que houvesse muito esforço dispendido na admissão de novos programadores.

Ainda segundo o artigo, apesar de todas as dificuldades com a ferramenta e a falta de uma metodologia mais formal que guiasse o desenvolvimento, a equipe aprovou seu uso frente ao resultado final. A experiência do grupo obtida com outros projetos do mesmo porte revelou que o uso da programação literária permitiu o uso real da documentação por causa de sua proximidade do código. Graças a essa característica a manutenção do programa foi extremamente facilitada, permitindo que os programadores alterassem códigos escritos por outros sem qualquer dificuldade. Além disso, a qualidade final do software ficou em níveis excelentes.

2.1.3 Literate Smalltalk Programming using Hypertext

Nesse trabalho [Ost93], Kasper Osterbye procurou determinar como e em que extensão hipertextos podem suportar a programação literária. Hipertextos já haviam sido sugeridos para gerenciar o processo de desenvolvimento de software como um todo. Seguindo essa linha, pensou-se em permitir que a programação literária fosse feita com o auxílio dos sistemas de hipertextos e estudar quão boa ou ruim seria tal abordagem.

O estudo teve início com uma pesquisa sobre os pontos fortes e os pontos fracos de WEB a partir dos quais seria analisada a utilização do hipertexto. Foram levantados, como positivos, os seguintes pontos:

- Documentação e programa são vistos como um único produto e não como artefatos distintos.
- Ao efetuar alguma mudança, o programador encontra o código junto com sua descrição no arquivo texto a ser editado.
- Permite que o programador desenvolva e apresente o código na ordem mais racional, sem se ater a uma determinada metodologia "top-down" ou "bottom-up".

As seguintes críticas foram feitas:

- Adiciona uma nova camada de complexidade no processo de desenvolvimento pois requer do programador o domínio das linguagens $\TeX$ e WEB.
- As referências cruzadas e os índices não estão disponíveis em tempo real, não auxiliando o programador.
- Não provê meios para documentar as demais fases do projeto, além da fase de codificação.

Com base nessas observações, foi criado um sistema de hipertexto para criação de programas em Smalltalk. O sistema permite a criação de ligações entre módulos de código e de documentação de uma maneira fácil, através de funcionalidades da interface. Dessa forma, o usuário cria seu programa literário não mais como uma linha sequencial de fatos, descrições e implementações mas como uma rede interligada de nós com estas finalidades. Usando as estruturas de nós e ligações do hipertexto, o usuário tem a sua disposição dois tipos básicos de ligações: referência e inclusão. Referências são usadas para correlacionar os trechos de documentação entre si e entre nós de código. Já as relações de inclusão são usadas para criar as ligações entre nós de código, estabelecendo, assim, as hierarquias.

As impressões obtidas desta implementação foram as seguintes:

- A estrutura de hipertexto estendeu WEB para permitir a inclusão de testes.
- Enfatizou os aspectos referentes às interrelações existentes entre os diversos trechos de código.
- A versão hipertextual do programa literário pode ser lida e consultada em tempo de projeto, auxiliando o programador em sua tarefa.
- Durante o processo de percorrimento do hipertexto, o usuário pode se perder dentro de sua estrutura (esta, porém, é uma característica comum a quase todos os sistemas de hipertexto).

Kasper Osterbye conclui o trabalho notando que algumas mudanças devem ser feitas para permitir o uso do sistema de hipertextos na programação literária. Entre elas temos: tornar a ferramenta independente de linguagem, implementar o conceito de "tangle" para que o código gerado possa ser passado a um compilador. Ainda assim, ficam algumas dúvidas quanto a seu uso:

- Não existe a lista de referências cruzadas e de índices geradas automaticamente por WEB.
- Existe o problema de se perder dentro do hipertexto.
- A facilidade de ligar trechos de código e de documentação não auxilia em nada a tarefa de manter coerente esta documentação.
- Os sistemas de hipertexto só podem ser usados "on-line", não disponibilizando nenhum tipo de documentação impressa do projeto.

2.1.4 Integrating a Hypertext Interface into a Syntax-Directed Programming Environment

É apresentada aqui [BR93] uma crítica ao modo tradicional de desenvolver programas: usar um editor para escrevê-los e um compilador, totalmente separado, para compilá-los. Segundo os autores essa forma de trabalhar traz consigo uma série de desvantagens. Uma delas é deixar a cargo do compilador a descoberta de erros de digitação e de sintaxe que poderiam ser acusados na hora de sua criação.

Essa situação pode ser revertida se usarmos os assim chamados editores dirigidos por sintaxe. O papel desses editores é apresentar ao programador modelos das estruturas de programação da linguagem usada, deixando a seu encargo apenas o preenchimento das estruturas com o código específico da aplicação. Com o uso desses modelos é possível identificar os eventuais erros na hora da escrita.

Além disso, tais editores podem ser estendidos para permitir a associação de comentários ao código e, desta forma, resolve-se um outro problema que é a documentação do programa. Essa abordagem sugere que o uso de hipertextos pode prover meios adequados para o acesso a essas informações. Pois, com esta organização, o programa pode se dividir em pequenos trechos de código que fazem referência às suas documentações. Com base nessas idéias este trabalho apresenta o projeto de um sistema de hipertexto integrado a um ambiente de programação dirigido por sintaxe.

O sistema proposto permite a programação em um sub-conjunto de Ada e usa modelos para as estruturas de controle da linguagem (tais como `IF`, `WHILE`, etc.) e para a sequenciação dos comandos. Não são usados modelos para expressões pois, apesar desses modelos garantirem expressões sintaticamente corretas, dificultam excessivamente sua digitação pelo programador. Tais modelos são implementados como árvores de sintaxe abstrata.

Sobre essa estrutura de representação, é possível, ainda, armazenar informações sobre o histórico do desenvolvimento do software. Esse histórico pode ser conhecido pois a edição do programa é feita sempre através do editor integrado ao sistema. Além disso, a informação pode ser facilmente guardada criando-se ramificações nas árvores de sintaxe abstrata dos modelos usados para a programação. Novas ramificações são criadas para cada um dos módulos componentes do programa que forem modificados. O critério para a criação de novas versões é o seguinte: uma vez que o foco de edição passe para um nível hierárquico superior e alguma alteração tenha sido feita junto a um módulo inferior, cria-se uma nova versão que representa todas as alterações feitas nos módulos inferiores. Essa política tenta, assim, minimizar o número de versões criadas.

Quanto à documentação, os autores não concordam com a visão de Knuth pois acham que os trechos de comentários atrapalham o fluxo de leitura do programa. Para resolver este problema e ainda manter a documentação integrada ao código, são utilizadas as características do hipertexto. Assim, um comentário pode ser ligado a um trecho de programa e invocado, a partir deste, por um simples clicar do mouse — sendo exibido em uma janela separada. Com essa arquitetura o programador passa ter somente o código exibido na tela, sendo os comentários acessados apenas quando desejado, o que, segundo os autores facilita a leitura do código.

Dois tipos de documentação podem ser incluídos no projeto: comentários dos modelos e comentários gerais. Os comentários associados aos modelos servem para descrever o significado do trecho de código em questão — exatamente como os comentários que costumamos incluir nos programas usando a sintaxe própria oferecida pela linguagem. Já os comentários gerais podem ser usados para documentar o projeto como um todo, incluíndo a especificação do sistema, as decisões de projeto, a arquitetura geral do código, etc. Como é o hipertexto que dá suporte a essas associações de comentários, fica natural a inclusão de referências cruzadas entre tais comentários. Permitindo, assim, que o programador crie ligações entre trechos que documentam partes da implementação com trechos correlatos da especificação do projeto. Este tipo de ligação favorece e facilita a integridade das documentações dos diversos níveis.

Além da documentação escrita, o sistema de hipertexto permite a inclusão de diagramas sem nenhuma dificuldade, visto que o editor disponibiliza meios para isso. A ausência dessa característica de inclusão de figuras foi sentida em ferramentas de programação literária. Com o uso de diagramas a explicação de complexas estruturas de dados são enormemente facilitadas.

Para manter o controle de versões, é necessário controlar, além do código, estes módulos de documentação. Isto é feito usando o mesmo processo utilizado no código: os comentários dos modelos são armazenados em árvores e novas ramificações são criadas a cada modificação feita. A partir daí são criadas as ligações entre as árvores para correlacionar as versões certas dos modelos aos seus respectivos trechos de comentários.

Como as árvores de código e documentação são independentes, é possível associar um único módulo de comentário com diversas versões de um modelo de código. Desta forma também se evita a proliferação de versões desnecessárias. Um processamento adicional é feito para que as ligações entre os comentários e os códigos de versões passadas sejam mantidas — desta forma, ao retornarmos para um ponto do passado, temos os comentários originais do código sendo exibido.

A estrutura de ligação entre comentários e modelos e entre os próprios comentários leva a uma complicada rede, ou seja, a um autêntico hipertexto. A possível desorientação do programador advinda desta estrutura de rede é minimizada, segundo os autores, pois os programadores estão acostumados a caminhar dentro dos programas e, como sempre há algum trecho de código sendo exibido, é possível uma re-orientação. Além disso, o esforço dispendido na criação desta rede também é minimizado, uma vez que grande parte é feita de modo automático, como as ligações inerentes entre um módulo e seus descendentes, entre usos de um identificador e o local de sua criação, etc. Desta forma, o maior trabalho deixado a cargo do programador é o de fazer as ligações corretas entre os comentários gerais e os comentários dos modelos.

O maior problema do sistema integrado de edição e hipertexto é, segundo os autores, o esforço adicional a que o programador se submete durante o uso do editor dirigido por sintaxe — muito embora esse esforço seja plenamente recompensado pelos ganhos obtidos, principalmente durante as fases de manutenção do projeto. Como os detalhes operacionais foram omitidos, não é possível uma análise mais detalhada deste aspecto.

2.1.5 Uma Ferramenta para Geração de Documentação de Sistemas de Software

Esse trabalho [dOB95] propõe uma arquitetura para documentação de programas. Na arquitetura proposta, a documentação deve ser introduzida no código pelo programador, seguindo um padrão pré-estabelecido de marcas. Com as marcas, a estrutura lógica do código pode ser explicitada e as diversas partes da documentação podem ser identificadas. Assim, por exemplo, o programador pode documentar uma classe, dando uma descrição funcional, descrevendo seus atributos, cada um de seus métodos etc.

Esse código fonte deve ser processado por uma ferramenta que faz uma análise sintática do mesmo, gerando como resultado uma árvore de sintaxe abstrata. Tal árvore é processada, pela mesma ferramenta, sendo instanciada em uma nova sintaxe — a sintaxe de uma segunda ferramenta que processa as informações obtidas, fazendo validação dos dados e gerando um arquivo de documentação no padrão HTML. A documentação final é acessada pelo usuário via uma ferramenta de hipertexto.

A ferramenta que implementa a arquietetura proposta é chamada **Documentu**. Em particular, o sistema é capaz de gerar documentação hipertextual para programas que não contenham marcas. Nesse caso, a documentação gerada apresenta apenas a estrutura do código — ou seja, o que se pode obter via análise sintática e transformação. São usados dois sistemas existentes que suportam as transformações descritas: Draco-PUC [LSF94] e Talisman [vS93].

Draco-PUC é um sistema de transformação entre domínios, onde um domínio é especificado via uma gramática estilo BNF. Tal gramática é passada ao subsistema **pargen** que gera um *parser* e um *unparser* para o domínio especificado. O parser faz a análise da entrada gerando uma árvore de sintaxe abstrata (DAST). Essa árvore de sintaxe abstrata pode, então, sofrer três tipos de processamento: ser mapeada, via transformações, para uma nova estrutura no mesmo domínio, para uma estrutura em outro domínio ou ser linearizada, pelo unparser, para a sintaxe concreta.

O sistema de transformações de Draco-PUC é totalmente configurado pelo usuário. Para especificar uma transformação, são fornecidos: o padrão a ser identificado, o padrão de substituição e diversas ações a serem tomadas em pontos pré-estabelecidos do processo de casamento do padrão. Tais ações podem destrutivas, quando trocam o padrão identificado por outro na própria fonte, ou podem podem ser não-destrutivas, quando apenas guardam informações em áreas de trabalho, gerando o processamento final em uma área de trabalho externa.

Talisman é uma ferramenta CASE baseada em um repositório de dados. A ferramenta trabalha com linguagens de definição que controlam seu comportamento. Uma linguagem de definição é composta por relações entre os elementos da base de dados e por fomulários.

Para trabalhar em um programa é necessário usar um formulário que permita alimentar a base de dados com as informações específicas do programa. Uma vez que a base esteja alimentada, formulários de validação podem ser executados para detectar possíveis erros de consistência entre os dados fornecidos. Além disso, formulários especiais, chamados *linearizadores*, podem ser usados para processar os dados do repositório gerando relátorios específicos.

As linguagens de definição determinam o comportamento de Talisman. Assim, para que possa ser configurável, novas linguagens podem ser criadas pelo usuário. Para tal, o usuário define as relações entre os elementos da base de dados, os formulários de entrada, os validadores e os linearizadores.

Documentu usa as ferramentas Draco-PUC e Talisman e implementa a arquitetura de documentação proposta para programas em C++. No sistema Draco-PUC foi utilizado um domínio C++ previamente existente e construído um domínio Talisman. Já em Talisman, uma nova linguagem de definição foi criada para dar significado à documentação obtida após o processamento feito por Draco-PUC. A linguagem criada define validadores para checar a completude das informações e a validade das referências cruzadas. Define, também, um linearizador que gera um documento hipertexto com base nas informações da base de dados.

Uma característica de Talisman que torna essa ligação possível é permitir que a base de dados seja alimentada por arquivos de importação. O domínio Talisman definido no sistema Draco-PUC reflete justamente a estrutura desses arquivos de importação de dados.

Dessa forma, o sistema **Documentu** trabalha da seguinte maneira: sobre os arquivos de implementação C++, com ou sem marcas de documentação, é executado o sistema Draco-PUC que transforma cada arquivo em uma árvore de sintaxe abstrata, instanciando-a, em seguida, para uma sintaxe concreta que reflete a estrutura dos arquivos de importação de dados de Talisman. Em seguida, Talisman é chamado, importando cada um dos arquivos gerados por Draco-PUC. Feita a alimentação dos dados no repositório, é feita uma checagem de consistência das informações obtidas. Qualquer inconsistência detectada nesse ponto é informada ao usuário. Após a validação dos dados, o linearizador entra em ação, gerando um documento hipertexto que reflete as informações obtidas no código.

Deve-se notar que o hipertexto gerado possui tanto o código quanto a documentação escrita pelo programador. Para tal, a estrutura do documento garante que apenas as informações mais relevantes são mostradas diretamente ao usuário, como os cabeçalhos de classes e as definições escritas na documentação. Os detalhes, tanto de documentação como o próprio código, podem ser acessados via links a partir do documento principal.

2.2 Ferramentas para Programação Literária

Nesta seção apresentamos as cinco ferramentas de programação literária analisadas.

2.2.1 CWEB

Ficha Técnica	
Nome	CWEB
Versão	3.0
Data	Junho, 1993
Título	CWEB System of Structured Documentation
Autor	Donald E. Knuth, Silvio Levy
Acesso	`ftp://labrea.stanford.edu`
Ambiente	C/C++ e TEX
Documentação	Manual do usuário (em TEX) incluído

Descrição

Como dito anteriormente, a metodologia de documentação de software proposta por Knuth foi implementada por ele para a criação de programas Pascal, como uma ferramenta chamada WEB. A demanda fez surgir, em um breve período de tempo, uma nova versão de WEB, nomeada CWEB, para a criação de programas na linguagem C. Esta versão foi implementada por Silvio Levy e manteve a mesma estrutura de WEB.

A ferramenta é composta por dois programas independentes: `cweave` e `ctangle`. O primeiro é responsável pela transformação do arquivo fonte escrito pelo programador em um arquivo no formato TEX que representa a documentação do sistema. O segundo faz a operação de "tangle", transformando o mesmo arquivo fonte do programador em um módulo C.

A estrutura do arquivo fonte escrito pelo programador é a mesma de WEB. À excessão dos comandos de macros e manipulação de strings (que já são tratados pelo preprocessador de C), todas as características de WEB foram mantidas aqui. Dessa forma, o documento gerado pela ferramenta apresenta o uma versão formatada do código, além de incluir referências cruzadas dos usos e das definições dos trechos de código e dos identificadores do programa.

Críticas

Esta ferramenta possui algumas restrições quanto a seu uso. Em primeiro lugar, o programador deve aprender uma nova linguagem, não trivial, que é a linguagem de comandos do CWEB. Desta forma, passa-se a trabalhar com quatro diferentes linguagens: a de programação (C/C++), a de documentação (português/inglês), a de formatação (TEX) e a de controle do CWEB. Além disso, o programador está amarrado a um determinado estilo de documentação: a estrutura descrita anteriormente exige a composição do documento em seções sub-dividas em três partes: uma inicial contendo uma descrição do material que se segue na seção, uma contendo definições de macros e a última contendo um (idealmente) pequeno trecho de código C que implementa a funcionalidade descrita no início da seção. Qualquer uma dessas partes pode ser vazia mas, ainda assim, as partes existem e o documento gerado refletirá essa divisão lógica do documento.

Há o problema da restrição das linguagens fonte: C ou C++. A experiência mostra que um grande projeto raramente usa apenas uma linguagem de desenvolvimento. Mesmo em sistemas triviais temos envolvidas, no mínimo, duas linguagens: a de programação (C, C++, Fortran...) e a do `make`. É comum usarmos, em sistemas gráficos de engenharia, uma linguagem de programação, uma linguagem de definição de interface (UIL de Motif [OSF91], LED de IUP [LdFG+96] etc.), uma linguagem de configuração da aplicação (Lua [IdFF96], por exemplo) e ainda a de definição do `makefile` — o que nos dá um total de quatro linguagens sendo usadas simultaneamente em um único projeto. Deste modo, restringir as linguagens de desenvolvimento restringe, em muito, as próprias aplicações a serem desenvolvidas. Além disso, CWEB nunca entenderá tanto de C ou C++ quanto o compilador usado (que, não necessariamente, entende *tudo* da linguagem), logo sempre haverá a possibilidade de erros por parte desta interpretação. Esse parser tem que ser feito para ser possível formatar o código e também para que os identificadores sejam postos no índice de referências cruzadas. Tais referências são interessantes, mas em grandes projetos talvez não seja adequado termos todos os identificadores listados (a ferramenta só não lista os identificadores de tamanho unitário) pois será excesso de informação.

Um ponto que não fica claro é quanto a formatação do código na documentação. Para um programador experiente, que escreve e lê o símbolo '`<=`' com o significado de menor ou igual, não vejo nenhum benefício (muito pelo contrário) em trocar tal símbolo por '$\leq$', ainda mais que o programador continuará escrevendo e lendo, no arquivo fonte, o símbolo original.

2.2.2 FWEB

Ficha Técnica	
Nome	FWEB
Versão	1.30a
Data	Junho, 1993
Título	FWEB System of Structured Software Design and Documentation
Autor	John A. Krommes
Acesso	`ftp://lyman.pppl.gov`
Ambiente	TeX e uma das linguagens: C, C++, Fortran ou Ratfor
Documentação	Manual do usuário incluído

Descrição

FWEB é uma ferramenta construída em cima de CWEB. Desta forma, todas as funcionalidades apresentadas estão presentes aqui. Dentre os melhoramentos, sua característica principal é a de permitir que o programador utilize outras linguagens além de C. São aceitas pela ferramenta as linguagens C, C++, Fortran, Ratfor e TeX. Além disso, as diversas linguagens aceitas podem ser usadas simultaneamente em um único documento/projeto.

Outra característica desta implementação é a portabilidade: FWEB roda na maior parte das plataformas existentes: VMS, PC e UNIX. De modo geral, FWEB rodará em praticamente todas as plataformas em que rodar o compilador C da GNU (gcc).

Críticas

Esta ferramenta já se mostra mais profissional, dada a preocupação em aceitar várias linguagens simultâneas e rodar em um grande número de plataformas. Porém, os problemas persistem: apesar do número de linguagens aceitas, continuamos presos a elas — uma limitação muito forte. A impressão formatada do código se manteve presente como uma qualidade, algo que ainda não temos consenso.

2.2.3 CLiP

Ficha Técnica	
Nome	CLiP
Versão	2.1
Data	Novembro, 1993
Título	Code from Literate Programs
Autor	Eric W. van Ammers, Mark R. Kramer
Acesso	`ftp://sun01.info.wau.nl`
Ambiente	qualquer
Documentação	Manual do usuário incluído

Descrição

A idéia de CLiP para programação literária é baseada no estilo de programação, ao invés do uso de comandos explícitos. Um dos objetivos principais é a simplicidade de uso. A ferramenta processa o arquivo do usuário, reconhecendo pseudo-comandos bem simples, embutidos em forma de comentários. Voltamos, então, para o caso típico de programação onde o programador só precisa manipular as linguagens diretamente envolvidas com o projeto — não há mais a necessidade de se aprender uma nova linguagem.

Outra característica fundamental em CLiP é a independência da linguagem de programação — CLiP, ao contrário das ferramentas expostas até aqui, não faz o parser do código, logo, o programador está livre para escolher as linguagens necessárias para o desenvolvimento. Além disso, CLiP também não prende o usuário a um determinado sistema de documentação, desde que este seja capaz de exportar uma versão ASCII do texto original.

A criação de um programa literário em CLiP deve ser feita da seguinte forma: O programador inicialmente escolhe a(s) linguagem(s) e o processador de texto a serem utilizados. A partir daí, começa o desenvolvimento do sistema segundo o paradigma da programação literária. O texto explicativo é posto em um formato especial que sugere o formato de comentários. Este texto gerado é a documentação definitiva: CLiP não possui o conceito de "weave" mas apenas o de "tangle". Assim, para gerar o código a ser compilado o usuário executa a ferramenta sobre uma versão (possivelmente o original) ASCII do documento escrito.

Críticas

CLiP peca por sua demasiada simplicidade. A ferramenta apenas extrai o código: nenhuma documentação extra é gerada. O usuário deve criar o documento já em seu formato definitivo, usando o formatador desejado. O processo a ser seguido é: o usuário digita o texto usando o formatador escolhido; para compilar, exporta o texto para formato ASCII; executa a ferramenta CLiP para extrair o código e, finalmente, pode compilá-lo: um processo muito extenso para ser realizado rotineiramente durante a fase de desenvolvimento do projeto e que, a princípio, implica na compilação total do código. Pelo modo de operar, CLiP também não é capaz de gerar referências cruzadas do código escrito, bem como qualquer tipo de índice.

2.2.4 FunnelWeb

Ficha Técnica	
Nome	FunnelWeb
Versão	3.0
Data	Maio, 1992
Título	FunnelWeb
Autor	Ross Williams
Acesso	`ftp://sirius.itd.adelaide.edu.au`
Ambiente	qualquer
Documentação	Manual do usuário incluído

Descrição

FunnelWeb é uma ferramenta de programação literária aos moldes de CWEB e FWEB. Ou seja, ela traz de volta toda a potencialidade das macros para o programador que havia sido posta de lado por CLiP. Porém, ao contrário daquelas, FunnelWeb é independente da linguagem de programação. Além disso, também é independente do sistema de formatação de texto no sentido de que o usuário não precisa se preocupar com este passo. Como em CWEB e FWEB, apenas cria-se o documento e, através da operação de "weave", obtém-se um arquivo no formato TEX contendo a documentação.

Desta forma, FunnelWeb se coloca em um meio termo: enquanto CWEB e FWEB exigiam o uso de duas linguagens adicionais (TEX e a de macros da ferramenta) e CLiP não exigia nenhuma, aqui apenas uma linguagem adicional (a de macros) é necessária.

Críticas

Por suas características, Funnelweb é uma ferramenta interessante porém ainda não ideal. Ainda existe o problema de forçar o programador a aprender uma nova linguagem, não trivial, de controle da ferramenta. Tal linguagem, assim como em CWEB e FWEB, dificulta a criação do código e até mesmo seu entendimento por ter uma sintaxe excessivamente baseada em símbolos. Isso pode ser visto em um pequeno programa criado a partir dos exemplos distribuídos junto com a ferramenta:

```
@O@<hello.c@>==@{@-
@<Include Files@>
@<Function@>
@<Main Program@>
@}

@$@<Main Program@>==@{@-
main()
{
 doit();
}
@}

@$@<Function@>==@{@-
```

```
void doit()
{
 @<Print@>
}@}

@$@<Print@>@M==@{@-
printf("Hello World!\n");@}

@$@<Include Files@>==@{@-
#include <stdio.h>
#include <stdlib.h>@}
```

Outro problema é que FunnelWeb gera a documentação em formato TEX mas não permite que o programador use os comandos do TEX para formatar o texto. Em particular, FW não permite o uso de caracteres acentuados o que exige um pré-processamento do texto e um pós-processamento do fonte TEX para que eles possam ser incluídos.

2.2.5 NOWEB

Ficha Técnica	
Nome	NOWEB
Versão	2.7a
Data	Março, 1995
Título	Simple, language-independent literate programming tool
Autor	Norman Ramsey
Acesso	`ftp://bellcore.com`
Ambiente	TEX/LATEX e qualquer linguagem de programação
Documentação	"Man pages" incluídas

Descrição

NOWEB, assim como CLiP e FunnelWeb, se destaca por não restringir as linguagens usadas no projeto. Também na linha de CLiP, procura ter a simplicidade como uma das características principais. Desta forma, não é exigido do programador o aprendizado de uma nova linguagem, ao contrário do observado em algumas das ferramentas já analisadas. Para exemplificar, reprogramamos o exemplo `hello.c` usando NOWEB:

```
\documentstyle[noweb]{article}
\begin{document}

<<hello.c>>=
<<Include Files>>
<<Function>>
<<Main Program>>
@

<<Main Program>>=
main()
```

```
{
 doit();
}
@

<<Function>>=
void doit()
{
 <<Print>>
}
@

<<Print>>=
printf("Hello World!\n");
@

<<Include Files>>=
#include <stdio.h>
#include <stdlib.h>
@

\end{document}
```

O código escrito é bem mais claro que o apresentado anteriormente mas aqui fica clara uma posição de dependência do NOWEB com relação ao sistema de documentação (no caso, LaTeX). Desta forma, exige-se do programador o uso, ao menos básico, deste sistema.

Críticas

NOWEB, assim como FunnelWeb, se mostra uma ferramenta bastante interessante. Não possuindo uma confusa linguagem de macros, permite que usuários possam utilizá-la apenas com os conhecimentos básicos de TeX e/ou LaTeX — sistemas de uso bastante difundido atualmente. Além disso, possui a óbvia vantagem de permitir o uso de todos os recursos oferecidos por esses sistemas.

Infelizmente, NOWEB peca em sua elegante implementação: poucos módulos binários compõem um núcleo de utilitários que, chamados por "scripts" do sistema Unix, fazem todo o processamento do documento criado pelo programador, gerando tanto os arquivos de código quanto os de documentação. Uma abordagem que privilegia em muito a organização do código mas deixa de lado todas as questões relativas à performance e à portabilidade. Outro ponto problemático é que o NOWEB usa demasiadamente as macros do TeX para gerar as referências cruzadas que são apresentadas ao usuário. Esse uso de macros, feito automaticamente, leva, por vezes, a aninhamentos de macros que extrapolam os limites de processamento do TeX, inviabilizando a geração do documento.

2.3 Ferramentas para Processamento de Textos

Diferentes tipos de ferramentas para processamento de textos existem hoje em dia. Desde macro-processadores simples como `m4` [Sun90], passando por `awk` [AWK], até poderosas linguagens de processamento de textos como `perl` [WS91]. Como veremos no capítulo 3, essa classe de programas pode auxiliar a construção de ferramentas de apoio à programação literária. Porém, uma série de características, que virão a ser enumeradas, deverão estar presentes no processador de textos para que este auxilie uma ferramenta.

Os processadores mais simples, como `m4`, não disponibilizam características indispensáveis. Em particular, esses programas não oferecem estruturas de dados versáteis para armazenagem e manipulação de trechos de texto, além de não possuírem facilidades para o processamento algorítmico dos dados lidos. `awk`, por sua vez, oferece uma linguagem na qual o processamento do texto lido pode ser feito. Porém, essa ferramenta é projetada para o processamento linha a linha, não facilitando o processamento de blocos de textos que se estendam por múltiplas linhas.

Os recursos desejados podem ser encontrados em `perl`, uma linguagem de processamento de textos voltada à extração de informações e geração de relatórios. Perl oferece, em sua linguagem, estruturas de dados capazes de armazenar grandes trechos de texto. Além disso, inclui comandos que permitem um processamento bastante satisfatório dos dados obtidos. Entretanto, `perl` possui uma complexidade muito grande de uso e instalação, além de estar disponível apenas para o sistema UNIX.

Capítulo 3

Um Framework para Programação Literária

Conforme discutido na introdução, a adoção da técnica de programação literária implica, por vezes, no desenvolvimento de uma ferramenta de suporte. A criação de novas ferramentas para projetos ou grupos específicos de usuários pode parecer, a princípio, contraditório, visto que há uma grande variedade de ferramentas já disponíveis. Porém, uma análise dos requisitos que uma ferramenta deve atender para ser utilizada em um projeto pode justificar esse procedimento. Tais requisitos, quando confrontados com as características identificadas na seção 2.1, nos mostram os fatores que inviabilizam o uso de uma ou outra ferramenta em um projeto específico:

- Dependência de plataforma — algumas ferramentas são dependentes de plataforma e podem não executar na plataforma sobre a qual o projeto deve ser desenvolvido.

- Dependência de linguagem — ferramentas podem ser específicas para processar programas em uma determinada linguagem que pode não ser a usada no projeto. Além disso, é comum o uso de mais de uma linguagem em um único projeto, o que agrava esse problema. Outro problema associado à dependência de linguagem de programação são possíveis erros de processamento da linguagem. Tal processamento é feito para que a ferramenta possa incluir, na documentação gerada, informações sobre usos e definições dos identificadores do programa. Quando se trata de uma linguagem como C++, cujos compiladores, hoje em dia, ainda não processam corretamente todas as estruturas da linguagem, as chances de erros por parte da ferramenta são grandes.

- Eficiência de uso — ao usar uma ferramenta de extração de código, adiciona-se um passo extra no processo normal de codificação/compilação/teste. A ferramenta deve ser eficiente para não aumentar esse tempo demasiadamente e, preferencialmente, deve poder ser adaptada aos processos automáticos como, por exemplo, os gerenciados por *makefiles*.

Além desses fatores, existem outros, mais fracos, mas que acabam por influenciar na decisão de se adotar ou não uma determinada ferramenta. Tais fatores são ditos mais fracos pois são de ordem estética ou pessoal e, dessa forma, uma mesma característica pode ser apontada como ruim por um programador e como boa por outro. Pode-se destacar:

- Formatação do código — a impressão formatada do código com palavras reservadas em negrito, nomes de identificadores em itálico, expressões matemáticas usando símbolos padrão, identação automática de blocos etc, pode não ser encarada como algo que facilite a leitura.

- Padrão fixo de formatação — algumas ferramentas fixam a formatação do texto não permitindo que o programador tenha liberdade para enfatizar determinados pontos, incluir figuras ilustrativas, fazer referências cruzadas entre partes do documento, etc.

- Dependência do formatador de texto — certas ferramentas geram a documentação para ser processada por um formatador de textos específico, não permitindo que o programador escolha o de sua conveniência, forçando-o, às vezes, a ter que aprender a usar esse formatador.

Diante desse quadro, fundamenta-se a proposta de criar um framework provedor dos serviços comuns às ferramentas de suporte à programação literária. Reforça-se aqui que a idéia desse trabalho não é apenas a de apresentar uma nova ferramenta mas, principalmente, prover meios que facilitem a criação de tais ferramentas.

Deve-se notar que o framework deve possibilitar a criação de uma ferramenta dotada de um sub-conjunto qualquer das características citadas. Para isso, é necessário não só identificar o núcleo comum de operações envolvidas, como também prover três fatores básicos: eficiência, portabilidade e flexibilidade.

A flexibilidade conseguida no framework será determinante da gama de ferramentas diferentes que poderão ser implementadas a partir deste. Como é justamente o conjunto de características que determina a adequação ou não de uma ferramenta a um caso, é fundamental que o framework seja o mais flexível possível para permitir o desenvolvimento de ferramentas tão diferentes quanto desejado. Por outro lado, a eficiência e a portabilidade são indispensáveis no framework para permitir que as ferramentas construídas possam, também, ser eficientes e portáteis.

3.1 Análise de Requisitos

A análise das ferramentas de suporte à programação literária permite avaliar como elas funcionam, segundo um ponto de visto abstrato. Essa seção, baseada no estudo apresentado na seção 2.2, discute que caracteríticas básicas são necessárias ao framework para que este possa ser usado como base para a construção de novas ferramentas.

Em um programa literário os trechos de código sempre podem ser diferenciados dos trechos de documentação, seja por marcas que os precedam, seja por estarem em uma seção específica do documento, seja por algum outro tipo de identificação. Esse tipo de diferenciação é feito pelo próprio programador usando a linguagem de controle da ferramenta. Como essas linguagens de controle são, essencialmente, linguagens de marcas, seu processamento se resume à identificar padrões, tomando ações semânticas associadas. A identificação de padrões, feita por macro-processamento por exemplo, resolve o primeiro passo no tratamento de um programa literário, que é a separação e nomeação das partes.

A nomeação de um trecho de código consiste na obtenção do nome segundo o qual o trecho pode ser referenciado para obtenção de seu conteúdo. Essa identificação deve ser feita e registrada para que referências futuras aos trechos possam ser resolvidas através da expansão do conteúdo do trecho em questão. Deve-se notar que, enquanto a obtenção desta informação pode ser resolvida com o uso de macro-processamento, a armazenagem deve ser feita em algum tipo de estrutura de dados. Em particular, o uso de tabelas associativas se mostra bastante adequado para a recuperação das informações desejadas.

Além desse processo de identificação e obtenção do conteúdo dos trechos de código, é necessário, em algum momento, identificar as referências feitas a estes. Dois são os motivos: a expansão do conteúdo para gerar o código desejado e a criação de referências cruzadas entre definições e usos dos trechos para gerar o documento final. A identificação, novamente, pode ser resolvida através de macro-processamento do conteúdo dos trechos de código, uma vez que tais referências são marcadas por algum tipo de padrão estabelecido. A geração do código também pode ser vista como resultado de macro-processamento, uma vez que representa a expansão em cadeia dos trechos referenciados. Montar as referências cruzadas, entretanto, implica em um tipo de processamento algorítmico que consulte os dados armazenados e registre usos e definições.

3.2 Desenho

Com base nas características observadas, o framework foi dividido em duas partes que atuam em diferentes níveis de abstração. Na base, com um menor nível de abstração, um núcleo em C++ implementa um macro-processador responsável pela leitura e escrita de arquivos, fazendo um processamento caracter a caracter da entrada. Sobre este núcleo, trabalha a linguagem de extensão Lua [IdFF96], que faz o processamento da fonte a nível de blocos de texto.

Essa estrutura visa conseguir o máximo de flexibilidade e eficiência para o framework. O uso de Lua como linguagem de definição e programação de macros traz a flexibilidade desejada; já a eficiência é obtida através do núcleo C++, projetado para tal fim. Segundo esse enfoque, apenas o núcleo, implementado em uma linguagem compilada e bastante otimizado, realiza as tarefas repetitivas, incluíndo o processamento caracter a caracter para a identificação de macros. Lua, uma linguagem interpretada, apenas controla o processamento, trabalhando com blocos de texto e sendo chamada somente quando as macros são identificadas na entrada, não prejudicando o desempenho. Não obstante, a portabilidade colocada como item fundamental é conseguida aqui. A implementação em C++ padrão, sem uso de templates, heranças virtuais e outras estruturas da linguagem que ainda não estão bem consolidadas, fazendo uso apenas de arquivos texto, garante sua compilação em praticamente todas as plataformas que disponibilizarem um compilador C++. O uso de Lua não compromete pois sua implementação foi feita em C padrão, sendo mais portável, até, que o núcleo.

Deve-se notar que macro-processadores comuns como m4 [Sun90] ou m5 [Sam92] não oferecem os recursos necessários para o tipo de processamento desejado. Tipicamente, a disponibilidade de estruturas de dados (tabelas, em particular) e a capacidade de processamento algorítmico não estarão entre os recursos oferecidos por estas ferramentas. Esse tipo de problema tem uma analogia na ferramenta NOWEB que usa as macros do TeX para codificar os algoritmos de identificação dos trechos de código e suas referências cruzadas — o que leva, às vezes, à extrapolação dos limites do TeX, inviabilizando a geração do documento, conforme apresentado na seção 2.2.5.

Na seção 3.3 é dada uma especificação funcional do framework enquanto na seção 3.4 é apresentada uma descrição de sua estrutura interna.

3.3 Especificação

O macro-processador trabalha lendo caracteres da entrada e repassando-os diretamente para a saída. Tanto a entrada como a saída são streams especificados pelo usuário e podem ser trocados a qualquer momento durante a operação. Esse processamento só é alterado quando uma macro é identificada na entrada. Nesse caso a leitura é interrompida, a ação associada à macro é executada e então a leitura é reestabelecida, sem que o texto da macro seja escrito na saída.

Por *stream* entende-se um objeto para o qual pode-se enviar caracteres ou do qual pode-se ler caracteres. Estão disponíveis para usuário streams para escrita e leitura de arquivos e de áreas de memória (*buffers*), bem como operações de concatenação de streams de leitura.

Por *macro* entende-se qualquer seqüência de caracteres registrada junto ao processador. Essa definição de macro implica na inexistência do conceito de macros com parâmetros, ao menos neste nível mais baixo do processador. Outro ponto importante é que também não existe o conceito de caracter ativo, ou seja, qualquer seqüência de caracteres pode representar uma macro, sem que haja um caracter especial para iniciá-la.

Essa ausência de um caracter especial para iniciar o nome das macros não significa qualquer limitação, uma vez que a aplicação que o desejar pode adotar um caracter comum para iniciar os nomes de suas macros. Já o mecanismo de parâmetros é substituido pelo conceito de ambientes de macros, como existente no macro-processador m5, associado a uma política de ativação adequada desses ambientes.

O conceito de ambientes é simples: um *ambiente* é conjunto de macros. A política de ativação também é simples: o processador possui uma pilha de ambientes na qual são postos os ambientes que estarão ativos durante a execução (processamento do texto de entrada). Essa estrutura permite que ações associadas às macros possam alterar a pilha de ambientes durante o processamento do texto para criar *contextos de execução* específicos para determinadas situações.

Assim, macros são registradas em ambientes e estes ambientes, por sua vez, são empilhados na máquina de execução do macro-processador. A idéia é que o texto sendo lido seja confrontado com as macros existentes no ambiente do topo da pilha em busca de um casamento e assim sucessivamente, descendo os níveis da pilha. Desta forma, empilhar um ambiente não significa eliminar o conjunto atual de macros ativas mas, sim, adicionar um novo conjunto ao universo de busca do processador.

Para permitir que o usuário possa trocar totalmente o ambiente de macros, ou seja, criar contextos diferentes de execução sem destruir o conteúdo da pilha, está previamente definido um ambiente `nil` que impede a busca de macros nos ambientes dos níveis inferiores a ele na pilha. Então, ao empilharmos o ambiente `nil`, todas as macros ativas deixam de ser vistas pelo processador e, empilhando-se sobre ele um ambiente comum, cria-se um contexto de execução no qual apenas as macros constantes desse novo ambiente serão usadas para processar a entrada. Essa funcionalidade é interessante para podermos, por exemplo, desativar o processamento de macros durante a leitura do parâmetro de uma macro, ou ainda para podermos implementar a tradicional macro *quote* que transporta seu parâmetro diretamente para a saída, sem qualquer tipo de processamento.

O critério usado para identificar uma macro na entrada é a igualdade do texto, não importando em que ambiente a macro está definida. Desta forma, se forem definidas as macros `a` e `ab`, a segunda nunca será identificada, mesmo que esteja registrada em um ambiente mais acima na pilha. Os critérios de decisão nos casos ambíguos são: nome mais extenso e macro declarada em um ambiente mais alto na pilha. Com esses critérios estão resolvidos os problemas associados à macros que sejam sufixo de outras macros e macros homônimas declaradas em ambientes diferentes[1].

Deve-se notar que a passagem de parâmetros para macros é resolvida com o uso de ambientes: por exemplo, uma macro `substitui(origem, destino)`, semelhante à macro `define` do m4, que troca todas as ocorrências do primeiro parâmetro pelo segundo, pode ser implementada com o uso três macros: '`substitui(`', '`,`' e '`)`'. O procedimento poderia ser: a macro '`substitui(`', registrada no ambiente normal de execução, ao ser encontrada, desviaria a saída do processador para um buffer auxiliar e empilharia um ambiente contendo as duas outras macros. A ação da macro '`,`' apenas desviaria a saída para um segundo buffer auxiliar. E a última macro, quando encontrada, desempilharia o ambiente do topo, restauraria a saída original e registraria no ambiente corrente uma macro cujo nome se encontra armazenado no primeiro buffer e cuja ação é escrever na saída o texto armazenado no segundo buffer.

A interface Lua do macro-processador permite o acesso normal às funcionalidades de criação de ambientes, registro de macros, inclusão e retirada de ambientes da pilha, além de poder disparar o processamento da entrada. O uso de streams, entretanto, não foi disponibilizado e uma nova interface de acesso aos buffers foi criada. Através dessa interface é possível criar buffers, concatená-los na entrada, enviá-los diretamente para a saída, convertê-los para strings Lua e vice-versa. Essa abstração deixa os programas Lua manipulando blocos de texto ao invés de caracteres, como feito via interface C++. Uma característica interessante dessa ligação é que um *programa* macro-processador (assim como m4 ou awk), pode ser facilmente criado a partir do framework.

3.4 Estrutura

Foi adotada uma estrutura modular para a construção do framework que se reflete na existência de camadas de funcionalidade bem definida que são construídas uma sobre a outra até a obtenção do resultado final. Seguindo essa estrutura, a primeira camada criada foi a responsável pela gerência de texto e será detalhada na seção 3.4.1. Após resolvida essa gerência, foram definidas as estruturas de ambientes de macros e a máquina de execução. Esses últimos serão descritos na seção 3.4.2 que cuida da gerência de macros.

A parte final do framework é sua ligação com Lua. O intuito, conforme exposto, é permitir a manipulação de macros por meio dessa linguagem de extensão. Os detalhes dessa ligação são abordados na seção 3.4.3.

3.4.1 Gerência de Texto

Por gerência de texto, no contexto do macro-processador, se entende ler caracteres de um repositório, escrever caracteres para um repositório e concatenar caracteres na frente de um repositório do qual se esteja lendo caracteres. O termo *repositório* tem, aqui, um significado não muito amplo, resumindo-se a arquivos texto e a áreas de memória mas podendo, se necessário, ser estendido para outras formas de armazenagem. Como dito anteriormente, as estruturas criadas para realizar tais tarefas foram os streams.

[1]Não é permitido armazenar duas macros iguais em um mesmo ambiente.

Streams

A idéia básica por trás dos *streams* é permitirmos que usuário leia ou escreva caracteres de uma forma homogênea, independente da origem ou do destino que estes caracteres venham a ter. Dessa forma, apenas dois tipos básicos devem criados: um que representa a classe dos objetos que lêem caracteres, e outro a classe dos objetos que escrevem caracteres.

A classe de stream de leitura poderia, a princípio, ser implementada com um único método virtual que retornasse um caracter, sendo esse método especializado em cada subclasse de acordo com a fonte dos caracteres. Porém, tal implementação implicaria em uma chamada de um método virtual a cada caracter a ser lido o que comprometeria o desempenho do programa. Uma estrutura muito mais eficaz pode ser obtida com o uso de uma área de memória e um contador do número de caracteres armazenados. Basta, então, usar essa área como um local de armazenamento temporário e prover um método inline que retorne os caracteres dali. Dessa forma, a leitura de um caracter de um stream tem, na maior parte das vezes, o mesmo custo de um acesso a um vetor. Obviamente, uma chamada a um método virtual é necessária para encher a área de memória, mas isso ocorrerá um número de vezes muito pequeno quando comparado ao número de caracteres lidos.

Para implementarmos essa política, a classe de stream de leitura deve possuir um ponteiro para os caracteres a serem lidos, o número de caracteres a ler e o método que será chamado para preencher a área de memória do stream com caracteres lidos da fonte. Esses dados devem ser de uso exclusivo dos objetos da classe, através do método de leitura de caracteres da fonte que, por sua vez, deverá ser codificada em cada uma das especializações da classe.

Note que não há uma área de memória no stream. Isso acontece porque, dependendo da fonte dos caracteres, uma especialização da classe pode usar uma área de dados da própria fonte. Dessa forma, fica a critério das especializações criar ou não uma área própria de memória para o armazenamento temporário.

A combinação de ponteiro para memória e posição foi pensada para permitir um acesso rápido aos caracteres — uma vez que desempenho é uma das diretrizes básicas do sistema. Isso porque com um ponteiro para um final do texto e mantendo o inteiro do número de caracteres a ler com valor negativo, podemos verificar se há mais caracteres a ler, indexar direto o vetor ou chamar a função de leitura, tudo de uma forma clara e eficaz.

Uma definição com um nível correto de abstração para os streams de escrita deve, assim como os de leitura, disponibilizar um ponteiro para o último caracter armazenado e um contador com valor negativo. Dessa forma, alcançamos a mesma eficiência de acesso conseguida durante o processo de leitura, por meio de um método inline que obtém caracteres. Analogamente, devemos dispor de um método que permita a escrita de mais caracteres após a área de memória usada pelo stream ter sido esgotada, ou seja, um método que, de alguma forma, esvazie a (possível) área de memória, permitindo que novos caracteres possam ser escritos. Novamente, a existência ou não dessa área de memória é deixada a critério das implementações de suas especializações.

A implementação dos streams de manipulação de arquivos, para ambos os casos, escrita e leitura, realmente cria uma área de memória para ser usada como buffer de escrita ou leitura dos dados trocados com o arquivo. Já os streams de acesso à memória utilizam estruturas criadas especificamente para esse fim, os buffers.

Buffers

Para solucionar o problema de armazenagem de caracteres em memória, recorreu-se ao conceito de buffers. Um *buffer* é um objeto capaz de armazenar uma quantidade indefinida de texto em memória. Um buffer deve ser composto, então, de uma área de memória para armazenagem, um indicador de seu tamanho, um indicador de sua ocupação e uma referência para um outro buffer que possa continuar a armazenar o texto caso sua memória se esgote. Os únicos métodos necessários em um buffer são o construtor, que deverá inicializar os atributos do objeto, e o destrutor, que deve liberar os demais buffers na lista. Nenhum outro será necessário pois este é um elemento passivo, sendo manipulado pelos streams exclusivamente.

Nessa construção foi observado um dos critérios iniciais para o desenvolvimento do macro-processador: a eficiência. Com essa estrutura de buffers, é possível implementar os métodos de leitura e escrita de caracteres como métodos inline dos streams que operam diretamente sobre a área de memória do buffer para obter ou armazenar os caracteres. Uma chamada de método (com late-binding) só será efetivamente necessária quando a área de memória do buffer estiver vazia em uma operação de leitura ou cheia em uma operação de escrita. No primeiro caso o método apenas ajusta os ponteiros internos do stream para se referenciar ao próximo buffer da lista, retornando o próximo caracter, ou, caso a lista tenha chegado ao seu final, retorna `EOF`. Já no segundo caso, um novo buffer deve ser criado, anexado à lista, e os ponteiros internos ajustados para se referenciar a esse novo buffer. Aliado à essa estrutura foi criado um pool de buffers para minimizar a criação dessas estruturas evitando, assim, constantes alocações e desalocações de memória.

Note que o conceito de buffers se adequa perfeitamente à aplicação do macro-processador uma vez que estes podem ser criados para armazenar partes da entrada para serem processadas a posteriori, podem guardar resultados de processamento que serão enviados à saída ou de volta à entrada, etc. Em particular, as "áreas auxiliares" exemplificadas na seção 3.3 podem ser implementadas como buffers.

3.4.2 Gerência de Macros

A gerência das macros está praticamente limitada aos ambientes. Neles serão definidas as macros com suas ações e neles a máquina de estados que busca as macros fará sua pesquisa. Como a busca de macros é a operação mais realizada pelo macro-processador (frente ao número de buscas, o número de registros de macros é desprezível), a estrutura dos ambientes deve facilitar tal tarefa.

Ambientes

Como um critério muito usado pelas aplicações é a adoção de um caracter especial que inicie o nome de todas as macros, podemos criar, como base da estrutura, um vetor de ponteiros para árvores de nomes. Desta forma, para saber se um dado caracter pode ou não iniciar o nome de uma macro, é necessário apenas indexar este vetor.

A árvore de nomes deve, então, guiar a busca caracter a caracter até que o nome de uma macro seja completamente identificado. Para implementar essa política, podemos colocar em cada nó da árvore:

- o caracter que indique o nome completado até ali.

- uma ação que, estando cadastrada, indique que uma macro foi encontrada e determina a tarefa a ser executada.
- uma lista de irmãos que implementa as alternativas de busca para caracteres diferentes.
- um ponteiro para a lista de filhos que permite a continuação da busca após a utilização do caracter corrente.
- sua altura, para que os critérios de desempate estabelecidos na especificação possam ser aplicados.

Nessa estrutura a inclusão de novas macros é simples, bastando percorrer a parcela de caminho já existente e expandir a parcela que faltar (caso falte), registrando a ação no último nó (caso já não exista uma ação cadastrada). A retirada também é simples, bastando percorrer o caminho recursivamente, retirar a ação do nó terminal e retornar apagando os nós que não tenham filhos e nem ações registradas (nós dedicados à macro retirada). De qualquer forma, tais estruturas não foram pensadas para facilitar a inserção ou a retirada de macros mas para facilitar a busca das macros a ser feita pelo processador na pilha de ambientes.

Pilha de Ambientes

Com relação a sua funcionalidade, a *pilha* é uma classe simples, só dispondo de métodos para especificar os streams de entrada e saída, empilhar e desempilhar ambientes, além de um método responsável por processar o stream de entrada, passando o resultado para o stream de saída. Dentre essas operações, a de maior importância em termos de desempenho é a operação de processamento: a busca de macros através dos ambientes ativos na pilha. Em termos de implementação, como a funcionalidade de empilhar um ambiente sugere que esta será uma operação feita raramente em muita profundidade, é possível limitar a profundidade máxima da pilha criando-a como um vetor de referências para ambientes.

Como a macro a ser identificada pode estar em qualquer nível, de qualquer um dos ambientes constantes da pilha, podemos implementar uma máquina de estados não determinística na pilha e no interior dos ambientes. Isso pode ser feito por um vetor de estados que acompanhe cada um dos possíveis caminhos, considerando-se todos os ambientes, frente à entrada. Ou seja, para cada caracter lido deve-se verificar cada um dos caminhos sendo percorridos para determinar se uma macro foi, ou não, encontrada com o uso do caracter corrente. Em caso positivo, a ação da macro deve ser executada e a máquina de estados deve ser reinicializada. Caso contrário o procedimento é mais complicado: deve-se verificar quais caminhos não podem mais ser percorridos e eliminá-los, além de incluir todos os caminhos possíveis que se iniciem no caracter em questão.

Deve-se notar que, adotando uma política de fila para a inserção de novos caminhos nesse vetor de estados e acessando os ambientes sempre a partir do topo da pilha e em direção à sua base, esta estrutura, por si só, nos garante o critério previamente estabelecido que diz que, se uma macro `A` for prefixo de uma `B`, `B` nunca será identificada. Além disso, nos garante também que, quando uma macro for definida em dois ambientes diferentes, a identificada será aquela que constar do ambiente mais alto na pilha. O último critério de desempate entre macros, que leva em consideração o comprimento dos nomes, já se encontra resolvido graças a altura dos nós na árvore que é guardada. Em termos de implementação vamos limitar, assim como na pilha, o número máximo de caminhos que poderão estar sendo percorridos em um mesmo instante.

3.4.3 Linguagem de Definição de Macros

Para permitir um acesso tão flexível quanto possível ao macro-processador, foi disponibilizada uma ligação deste à Lua [IdFF96]. Lua é uma linguagem embutida usada, tipicamente, para configurar aplicações. Seu uso será destinado a facilitar a criação de macros por parte do usuário, além de permitir a codificação de funções responsáveis pelas ações a serem executadas por estas macros.

A função deste módulo é, então, fornecer uma *API* de acesso aos módulos existentes de gerência de texto e macros. Porém, a forma com que esses serviços serão acessados deve ser simplificada e adaptada para o uso pelo usuário final. Para tal, uma pilha fixa será utilizada como base de todas as operações e o código Lua não terá acesso aos streams — somente aos buffers. Além disso, as operações disponíveis para a manipulação dos buffers serão mais especializadas: buffers poderão ser concatenados diretamente na entrada da máquina de execução, poderão ser escritos diretamente na saída ou, ainda, poderão ser postos como saída corrente da máquina.

Como Lua lida apenas com buffers, não tendo acesso aos streams, foram disponibilizadas funções utilitárias para criar e destruir buffers além de funções para convertê-los em strings e vice-versa. Já para a manipulação de macros foram criadas funções para criar e destruir ambientes, registrar macros (função esta que recebe uma *função Lua* como ação a ser executada pela macro), empilhar e desempilhar ambientes, e ativar a máquina de processamento. Além dessas funções, também foram criadas as já mencionadas para concatenar um buffer na entrada da máquina, copiar um buffer para a saída, colocar um buffer como saída corrente da máquina e restaurar a saída anterior (alterada pela colocação de um buffer).

O ponto alto dessa ligação é permitir que programas Lua tenham total acesso e controle sobre o núcleo do macro-processador. Foge, porém, ao escopo deste trabalho fornecer uma descrição detalhada da linguagem de extensão.

Capítulo 4

A Ferramenta nome para Programação Literária

Esse capítulo apresenta a ferramenta **nome**, de suporte à programação literária, desenvolvida sobre o framework descrito no capítulo 3. O desenvolvimento de **nome** atende a dois propósitos: testar se o framework atingiu os objetivos propostos e disponibilizar uma nova ferramenta.

nome compõe-se de dois programas: `code` e `doc`. Esses programas são responsáveis, respectivamente, pela extração do código e pela geração da documentação a partir dos fontes criados pelo programador. As características desses programas e do programa literário a ser processado por eles foram estabelecidas tomando-se por base a análise feita na seção 2.2. Foi estabelecido que a ferramenta deveria ser:

- Independente da linguagem de programação utilizada no projeto de modo a permitir que várias linguagens sejam usadas simultaneamente (make, C, LED, Lua, etc).
- Portável para poder ser usada em qualquer ambiente de desenvolvimento.
- Eficiente, pois está incluíndo um passo adicional no processo normal de desenvolvimento: codificação/compilação/teste.

Essas caracteríticas foram assim definidas para eliminar os três primeiros fatores identificados no capítulo 3 como possíveis determinantes da rejeição de uma ferramenta em um projeto, a saber: dependência de linguagem, dependência de plataforma e ineficiência. Deste três fatores, o primeiro é pura decisão de projeto. O segundo foi resolvido através de uma implementação em C++ e Lua como a do framework, sem usar artifícios ainda não bem consolidados da linguagem C++. O terceiro foi anulado pela eficiência do próprio framework.

Quanto aos fatores ditos mais fracos, as opções tomadas foram as seguintes:

- Ausência de formatação de código — uma vez que a ferramenta deve ser independente da linguagem de programação utilizada. Deve-se notar que a impressão formatada do código implica em uma de duas alternativas: ou a ferramenta conhece a priori a linguagem a ser utilizada ou é fornecida uma descrição em tempo de exe-cução. A fim de manter independência, as linguagens usadas não são reconhecidas e o código não pode ser formatado.

- Nomes de trechos com identificação — os trechos de código são identificados automaticamente para permitir a criação de referências cruzadas apontando os usos e as definições dos trechos. A presença das referências cruzadas facilita a leitura da documentação uma vez que essa leitura, na maior parte das vezes, visa partes específicas, ao invés de ser uma leitura completa seqüêncial.

- Liberdade de formatação — o programador está livre para formatar o texto da forma que melhor lhe convier. A ferramenta inclui formatações fixas apenas para os trechos de código, incluíndo o nome do trecho, seu conteúdo e as informações geradas automaticamente (referências cruzadas de usos e definições). A formatação dos trechos de documentação é definida pelo programador.

- Simplicidade da linguagem de controle — as seqüências de controle da ferramenta são poucas e simples, evitando o overhead do aprendizado de uma nova linguagem para o programador.

O processador LaTeX foi adotado para a geração da documentação. O uso de um processador de textos fixo foi feito considerando-se que: esse programa é bastante difundido na comunidade de informática; permite que todo o controle da formatação fique a seu encargo o que levou à simplificação da linguagem de controle de **nome**; disponibiliza meios de criação automática da identificação dos trechos e de suas referências cruzadas. Ao mesmo tempo que traz essas vantagens, o uso do LaTeX exige que o programador domine ao menos seus princípios básicos para poder criar a formatação do texto, o que, estando o uso do processador bem difundido, não chega a ser um empecilho uma grande parte das vezes.

Como o controle da formatação fica a cargo do LaTeX, apenas os comandos de controle dos trechos de código e de documentação tiveram que ser criados. Desta forma, a linguagem de controle de **nome** permaneceu extremamente simples: apenas oito seqüências de controle foram necessárias. A linguagem criada se assemelha bastante à linguagem utilizada pelo NOWEB.

As referências cruzadas exibidas na documentação, conforme mencionado, são criadas usando o sistema de referências do próprio LaTeX. Com isso temos o resultado desejado, com impressão do número da página e letra na identificação do trecho, sem o uso de pesadas macros do TeX, como feito pelo NOWEB. O exemplo abaixo simula um trecho de código, de nome '*Inicializa máquina de estados*', que é o primeiro trecho de código definido na página 23 — daí sua identificação *23a*. A observação ao término da definição indica que esse trecho é referenciado dentro do segundo trecho definido na página 35.

$\langle$*Inicializa máquina de estados 23a*$\rangle \equiv$

```
{
  for (int nivel=0; nivel<NUM_NIVEIS; nivel++)
    maquina[nivel] = 0;
}
```

Este trecho é usado em 35b

Foi disponibilizada uma opção para gerar informações de posicionamento das linhas no arquivo fonte. Essa opção tem como default a criação de diretivas segundo o padrão usado pelo pré-processador de C, mas pode ser alterado para gerar informações segundo qualquer padrão que inclua número de linha e/ou nome do arquivo de origem, além de qualquer texto fixo. Essa característica é essencial para o desenvolvimento, uma vez que todas as edições são feitas no arquivo fonte: referências feitas por um depurador, por exemplo, aos arquivos de código gerados automaticamente não fariam sentido algum para o usuário.

O texto escrito pelo usuário não precisa seguir qualquer tipo de estrutura previamente determinada, mas apenas a estrutura básica que o divide em trechos de documentação e trechos de código. As seqüências de controle se destinam a identificar e marcar o início e o fim dos trechos de código (tudo o que estiver fora deles será visto como trecho de documentação), além de permitir referências a outros trechos para expansão.

O usuário fica livre para estruturar seu texto da forma que melhor lhe convier, inclusive com a separação em vários arquivos. Essa separação em arquivos conta, também, com um suporte à modularidade do código. O objetivo desse suporte à modularidade é criar um mecanismo que permita referências cruzadas entre trechos de arquivos diferentes. Dessa forma, o usuário pode definir trechos em um arquivo e usá-los em outro, obtendo expansões de um mesmo trecho em diferentes arquivos. Uma política de controle define que trechos são exportados pelo módulo e quais não são, estabelecendo, assim, o conceito de modularidade.

Todas essas características da ferramenta são controladas através das nove seqüências de controle que se seguem:

`<<` — dentro de um trecho de documentação, termina este, iniciando um trecho de código cuja identificação se segue.

`*<<` — dentro de um trecho de documentação, termina este, iniciando um trecho de código exportado cuja identificação se segue.

`>>=` — termina a identificação do trecho de código cujo conteúdo se segue.

`@` — dentro de um trecho de código e sendo o último caracter de uma linha, termina o trecho de código sendo definido e inicia um trecho de documentação.

`<<` — dentro de um trecho de código, inicia a identificação de um trecho sendo referenciado.

`>>` — dentro de um trecho de código, termina a identificação de um trecho sendo referenciado.

`@<<` — permite a inclusão da seqüência `<<` sem efeito de controle.

`@*<<` — permite a inclusão da seqüência `*<<` sem efeito de controle.

`@@` — permite a inclusão do caracter `@` sem efeito de controle.

Usos dessas seqüências em exemplos reais de programação podem ser encontrados nos apêndices.

Assim como a ferramenta desenvolvida por Knuth, **nome** adiciona um passo no processo normal de desenvolvimento. Diferencia-se, entretanto, em aspectos de múltiplas linguagens, suporte à modularidade e processador de textos (LaTeX). Um esquema de uso análogo ao mostrado na figura 2.1, incluíndo tais diferenças, pode ser visualizado na figura 4.1

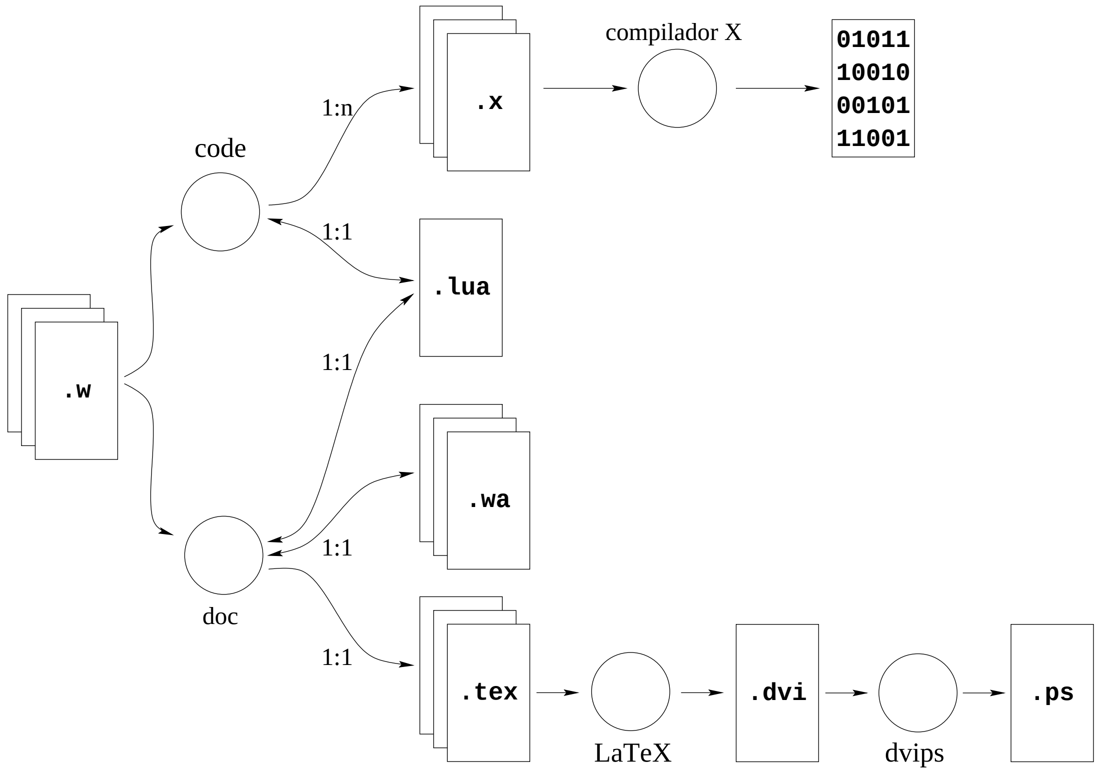

Figura 4.1: Esquema de uso da ferramenta **nome**.

A próxima seção aprensenta a solução adotada para dar suporte ao desenvolvimento modular com a ferramenta. Já as duas últimas seções se encarregam de dar uma descrição detalhada da arquitetura de funcionamento dos programas: a seção 4.2 trata do programa de extração de código, enquanto a seção 4.3 aborda o programa responsável pela criação da documentação.

4.1 Suporte ao Desenvolvimento Modular

Para permitir uma otimização da separação do fonte em arquivos, foi disponibilizado um suporte à modularidade do código. A idéia é permitir que o programador defina trechos de código em um módulo e os use em outros módulos além do de origem. Uma vez definidos tais trechos, ditos trechos exportados, outros módulos podem fazer referências a estes que resultarão na expansão do respectivo código. Deve-se notar que um módulo não pode fazer referência a qualquer trecho de outro módulo mas somente aos exportados. Por outro lado, durante o processo de expansão, o trecho exportado pode fazer referência a todos os trechos locais de seu módulo e aos trechos exportados pelos demais módulos do projeto.

Para que esse conceito possa ser posto em prática é necessário criar um repositório de dados comum contendo informações sobre os trechos exportados por cada um dos módulos componentes do projeto. Esse repositório foi incluído em um arquivo de configuração cujo objetivo é armazenar informações gerais sobre o projeto. As informações relevantes para o mecanismo de modularidade são o nome do trecho exportado e o arquivo no qual ele é definido. Com essas informações é possível, ao processar um trecho que referencia um trecho exportado, processar o arquivo de origem para obter o resultado da expansão do trecho desejado.

A escolha da arquivo de configuração foi feita pois atende ao objetivo e é fácil de ser mantido. A cada execução de **nome** sobre um módulo, o arquivo de configuração é lido e, após feito o processamento completo do módulo, o arquivo é gravado atualizado com referência ao módulo processado. Dessa forma, na pior das hipóteses, pode ser necessário que cada um dos módulos componentes do projeto sejam processados uma vez para gerar uma versão do arquivo de configuração contendo informações completas sobre todos os trechos exportados do projeto. Assim, uma segunda execução, ainda na pior das hipóteses, seria capaz de gerar os códigos completos de cada um dos módulos.

Essa escolha de armazenagem, associando apenas o nome do trecho exportado ao seu módulo de origem, faz com que não hajam problemas com versões de arquivos. Se a associação feita fosse nome do trecho versus conteúdo, seria importante definir uma política de atualização do arquivo de configuração para evitar que uma versão desatualizada do trecho exportado fosse utilizada em uma expansão. Com a solução adotada, o arquivo em questão é processado na hora em que o trecho deve ser expandido, levando sempre à versão mais atualizada deste.

Esse arquivo de configuração é criado e mantido automaticamente por **nome**, que dispõe de parâmetros de execução para a especificação do arquivo a ser usado. Com isso é possível criar projetos diferentes e mantê-los sob a estrutura de diretórios que melhor convier. Outros dados que podem constar deste arquivo é uma lista completa dos módulos componentes do projeto para que **nome** possa gerar uma versão completa do código e da documentação de uma forma mais direta.

4.2 Code

O programa code, utilizado para extrair o código do programa literário, foi codificado em C++ e Lua. A maior parte das funcionalidades estão presentes no script Lua, sendo o código C++ responsável apenas pelo parser da linha de comando e pela prestação de alguns poucos serviços ao código Lua — é o script Lua que cria macros e ambientes, além de controlar o processamento da entrada.

Inicialmente, a saída do processador é direcionada para um stream especial que não armazena a informação recebida. Esse recurso é utilizado pois, durante o processamento do arquivo fonte, apenas os trechos de código (doravante chamados apenas *trechos*) devem ser armazenados, sendo os blocos de documentação ignorados.

Para iniciar o processamento, empilha-se um ambiente contendo uma macro que representa o início da identificação de um trecho. Essa macro, quando encontrada, desvia a saída para um buffer auxiliar, empilha um ambiente contendo a macro que representa o fim da identificação do trecho e segue com o processamento. A ação correspondente à macro de fim de identificação executa as seguintes tarefas: transformar o buffer auxiliar em uma string que identifica o trecho (seu nome), registrar em um ambiente especial uma macro com esse nome, desviar a saída para um segundo buffer auxiliar, desempilhar o ambiente do topo da pilha, empilhar um ambiente contendo a macro de final de trecho e voltar ao processamento. A macro de final de trecho, ao ser encontrada, registra em uma tabela associativa o trecho lido (fazendo a associação $<$*nome*:*buffer*$>$), desempilha o ambiente que a contém, restaura a saída e continua o processamento.

É importante notar que, ao término do processamento do arquivo fonte, estarão registrados nesta tabela associativa os conteúdos de todos os trechos de código definidos pelo usuário, acessáveis pelos seus nomes. Essa tabela chama-se *tabela de trechos*. Além disso, o ambiente especial mencionado possuirá uma macro para cada nome de trecho. Esse ambiente chama-se *ambiente de macros*. Essas duas estruturas de dados são, na verdade, guardadas em tabelas cuja chave de recuperação é o nome do arquivo corrente. Dessa forma, contextos diferentes podem ser criados para arquivos diferentes sem que haja qualquer correlação entre esses.

O segundo passo consiste em gerar o código desejado. Para tal, o trecho raiz é expandido na entrada, o ambiente de macros referente ao arquivo principal (arquivo fornecido pelo usuário) é posto como único ambiente da pilha, a saída original é restaurada e o processamento reativado. A partir desse ponto, referências feitas a outros trechos causarão a execução de macros cuja ação apenas expande o código correspondente na entrada. Dessa forma todos os trechos necessários são expandidos na ordem correta para gerar o arquivo final.

Fatores complicadores desse processo são a inclusão de informações de posicionamento dos trechos nos arquivos fonte e o uso de trechos externos, definidos em outros módulos. Para resolver o posicionamento, existem macros associadas à mudança de linha nos diversos ambientes que mantém um contador da linha corrente no arquivo fonte. Com o uso deste contador, a informação de posicionamento é registrada na tabela de trechos, juntamente com o conteúdo do trecho, sendo expandida nos limites dos trechos.

Trechos externos, conforme dito anteriormente, são resolvidos através de um arquivo de configuração que contém a lista dos trechos externos e os módulos que os definem. Esse arquivo é processado no inicío da execução (antes de processar o arquivo principal) sendo os trechos lá existentes postos na tabela de trechos referente ao arquivo principal. Durante o processamento para gerar a saída, se um determinado trecho, existente na tabela, não se encontra acompanhado de seu conteúdo mas apenas do nome do módulo que o define, faz-se o processamento do arquivo fonte em questão. Ao término desse processamento, a função de expansão de trechos pode, então, identificar que o trecho a ser expandido tem seu conteúdo registrado na tabela de trechos do arquivo que o contém. Assim, usando essa tabela e o ambiente de macros também desse arquivo, faz-se a expansão do trecho na saída.

Deve-se observar que com a solução adotada não são criados problemas com a versão dos arquivos utilizados, uma vez que o arquivo fonte é processado no momento em que o trecho é requisitado. Outro ponto importante é que o trecho externo, quando expandido, tem acesso a todos os trechos de seu módulo de origem e a todos os trechos externos do projeto, mas somente a eles, ou seja, em momento algum é permitida a referência a um trecho não exportado de um módulo por um trecho qualquer de outro módulo. Com isso impõe-se as regras desejadas para modularidade. Ao término da execução a tabela de trechos externos, possivelmente modificada frente ao processamento do(s) arquivo(s), é reescrita no arquivo de configuração.

4.3 Doc

Assim como no programa code, a maior parte das funcionalidades de doc estão no script Lua. Esse script controla todo o processamento do fonte, cabendo ao código C++ apenas o parser da linha de comando e a ativação do script.

Uma vez ativo, o código Lua trabalha transferindo o texto lido do arquivo fonte diretamente para a saída até que um trecho de código seja identificado. Tal identificação ocorre de forma análoga à descrita para o programa code. Obtida a identificação de um trecho, esta é enviada para a saída juntamente com os comandos LaTeX responsáveis por sua formatação e sua rotulação. Também são geradas marcas do LaTeX para permitir futuras referências a esse trecho.

Por *rotulação* entende-se uma identificação auxiliar anexada ao nome do trecho composta pelo número da página seguida de uma letra que o diferencie dos demais trechos da mesma página (três trechos existentes na página 25 seriam rotulados como 25a, 25b e 25c). Para fazer essa rotulação, foi criado um contador que o próprio TeX se encarrega de atualizar a cada mudança de página.

Uma vez escrita a identificação completa do trecho, cada linha é processada, sendo enviada para a saída juntamente com comandos de formatação do LaTeX. Tais comandos são responsáveis pela exibição do código em fonte courier e pela não interpretação pelo LaTeX desses caracteres.

Ao término do trecho, devem ser postas informações pertinentes, como trecho raiz, continuações e usos. Por *trecho raiz* entende-se um trecho que não é referenciado por qualquer outro do documento, ou seja, seu único uso é ser expandido para gerar um arquivo de código. O conceito de *continuação* permite que um trecho de código seja escrito em partes. Para isso, basta que dois ou mais trechos de código possuam o mesmo nome, sendo o resultado de sua expansão igual à concatenação do conteúdo de cada trecho. Em termos de documentação, é usado um símbolo de adição nos nomes das continuações para passar a idéia de que o trecho em questão está continuando um trecho previamente definido. As informações relevantes, nesse caso, são, no trecho inicial, indicar em quais trechos ele é continuado e, nos trechos subseqüêntes, indicar qual trecho está sendo continuado. Quanto ao uso dos trechos, é importante informar quais trechos se referenciam ao trecho corrente.

É importante observar que, muitas vezes, as informações necessárias ao término de um trecho não serão, ainda, conhecidas. Por exemplo, ao processar o primeiro trecho de uma série de continuações, deve-se explicitar todos os trechos da série (ou seja, todos os trechos no qual este primeiro trecho será continuado) porém tais trechos ainda não foram processados e essa informação ainda não existe. Problema análogo pode exitir quanto às informações sobre os usos de um trecho, visto que o usos futuros ainda não terão sido detectados.

Para resolver tais problemas foi adotada uma solução como a dada pelo TeX. Um arquivo auxiliar para cada módulo processado contém todas as informações necessárias: se o trecho é continuado ou não, quais trechos compõem sua continuação e quais trechos o referenciam. Este arquivo auxiliar é lido antes do processamento do arquivo fonte e escrito, atualizado, ao término deste. Dessa forma, uma segunda execução pode ser necessária para obter a documentação correta. Nesse caso o próprio programa identifica a necessidade de novo processamento e indica ao usuário.

A montagem dessa estrutura é feita com o uso das tabelas associativas de Lua. Dessa forma, uma tabela indexada pelo nome do trecho armazena as informações necessárias como número do trecho (usado internamente), número de continuações, informações de cada continuação e trechos que o referenciam. Essa tabela é montada aos poucos, com informações recolhidas na primeira vez que um trecho é declarado, nas vezes em que é referenciado e nas declarações subseqüêntes de suas continuações.

Capítulo 5

Conclusões

As análises feitas no capítulo 2 nos mostram que a técnica de programação literária parece ser relevante para a documentação de programas. Entretanto, a falta de adequação das ferramentas existentes a particulares projetos de desenvolvimento, seja por portabilidade, aceitação das linguagens usadas, preferências pessoais ou outras quaisquer, faz com que seja necessário, por vezes, o desenvolvimento de uma ferramenta específica. Nós imaginamos que a criação de uma ferramenta pode realmente ser necessária para um grupo mas não para cada projeto em particular. Dessa forma, o esforço pode vir a ser recompensado. Em tal contexto, o desenvolvimento de um framework de suporte à construção de ferramentas de programação literária é bem recebido pois aumenta a viabilidade (custo/benefício) da construção de uma ferramenta específica.

O framework foi desenvolvido para permitir a construção de ferramentas de forma genérica, ou seja, nenhum conceito de programação literária está previamente colocado lá. Apenas as tarefas comuns, de macro-processamento apoiado por codificação algorítmica e estruturas de dados, compõem este núcleo. Outras preocupações tomadas no desenvolvimento foram flexibilidade, eficiência e portabilidade.

A flexibilidade do framework se apóia em sua arquitetura, e na flexibilidade da linguagem Lua [IdFF96]. Apenas as operações básicas de manipulação de texto estão implementadas em C++, formando uma biblioteca sobre a qual pequenos programas em Lua definem o formato final da ferramenta.

A eficiência, almejada durante todo o projeto, foi alcançada. A ferramenta **nome**, desenvolvida sobre o framework, apresenta um desempenho bastante superior ao do NOWEB (ver Figura 5.1). A documentação LaTeX, por exemplo, é gerada em menos de 15% do tempo necessário para o próprio LaTeX tratar o texto, resultando portanto em um custo adicional pequeno para o desenvolvedor.

A portabilidade foi obtida com uma implementação em C++ evitando-se o uso de facilidades ainda não bem implementadas, como tratamento de excessões, *templates*, e herança virtual. Além disso, todas as bibliotecas usadas fazem parte do padrão ANSI. A biblioteca de implementação da linguagem Lua também é bastante portátil, e vem sendo usada em ambientes DOS, Windows 3.x, Windows NT, Unix (SunOS, Linux, IBM/AIX, Silicon/IRIX, etc) e Macintosh. Os programas code e doc já foram instalados, sem mudanças no código, nas plataformas Unix (SunOS, IBM/AIX, Linux, Silicon/IRIX, DEC/ULTRIX), MS-DOS e Windows 3.x.

As características de flexibilidade, eficiência e portabilidade apresentadas pelo framework sugerem que este pode ser utilizado para a construção de novas ferramentas

Comparação entre Ferramentas		
	code & doc	NOWEB
Extração de código	11.5s	13.1s
Geração da documentação	8.0s	59.0s

Geração do Executável			
	code	make	total
Compilação	11.5s	50.0s	61.5s

Geração da Documentação			
	doc	LaTeX	total
Processamento	8.0s	57.6s	65.6s

Figura 5.1: Medidas de desempenho de **nome** para um fonte de 33.000 linhas de código.

de apoio à programação literária bem como outras aplicações. Em particular, macro-processadores para usuários finais podem ser escritos com grande facilidade.

O conceito de programação literária foi explorado durante todo o desenvolvimento desse trabalho. Inicialmente, o framework foi desenvolvido como um programa literário, usando a ferramenta NOWEB [Ram92] como suporte. Tal desenvolvimento levou à uma documentação coesa e precisa.

Seguindo a mesma linha, a ferramenta **nome** teve seu desenvolvimento baseado na filosofia de programação literária. Os scripts de configuração do framework que implementam a ferramenta foram repetidas vezes alterados para testar diferentes abordagens. Tais alterações acarretaram em mudanças correspondentes na documentação, fazendo com que a escrita da documentação fosse refeita diversas vezes. Um esforço muito grande foi despendido inicialmente para que código e documentação caminhassem juntos. Porém, após diversas mudanças, a documentação foi posta de lado para que a experimentação pudesse ser feita de forma mais livre e rápida. Assim, a documentação acabou por ser redigida somente após o desenvolvimento da versão definitiva do código que implementa os scripts. Essa experiência mostrou que a técnica de programação literária acaba por exigir demais em um processo de programação exploratória.

Após a disponibilização da primeira versão de **nome**, o projeto como um todo, englobando framework e ferramenta, foi traduzido para **nome**. Dessa forma, os próprios programas `code` e `doc` são utilizados para continuar o desenvolvimento. Além disso, vários outros projetos estão atualmente sendo desenvolvidos sob o paradigma da programação literária, tendo **nome** como ferramenta de suporte.

Apêndice A

Exemplo de Transformação

Nesse apêndice é mostrado um exemplo de utilização da técnica de programação literária. O objetivo desse exemplo é mostrar como o código fonte escrito pelo programador é transformado, levando a um arquivo de código e um de documentação. Dessa forma, o leitor poderá visualizar o resultado dessas transformações, além do resultado final da documentação já processada pelo LaTeX.

Na seção A.1 é mostrado o fonte criado pelo programador. Na seção A.2 se encontra o resultado da transformação do fonte pelo processador `code`. O resultado da transformação realizada por `doc` é mostrado na seção A.3. Já na seção A.4, é apresentado o resultado final do processamento do arquivo de documentação pelo processador LaTeX.

O exemplo trata-se de um trecho da implementação do framework apresentado neste trabalho. Mais precisamente, o trecho em questão versa sobre a gerência de textos.

A.1 Código Fonte

O texto apresentado a seguir, em fonte de máquina, é o texto criado pelo programador.

```
A gerência do texto pode ser dividida em dois problemas principais a serem
 resolvidos: a armazenagem e a manipulação.
Quando o texto tem sua origem, digamos, em arquivos, não precisamos nos
 preocupar com armazenagem mas apenas com sua manipulação.
A mesma situação se repete quando se trata de texto a ser enviado para um
 arquivo, ou até mesmo quando se trata de algum outro elemento de armazenagem.
Porém, quando este não for o caso, devemos disponibilizar ao usuário uma forma
 de armazenagem.
Isso pode ser conseguido através de objetos chamados [[buffers]].
Tais objetos têm, como único objetivo, armazenar texto para manipulação.

Já o processo de manipulação deve ser todo criado pois, não importando a fonte
 (ou o destino) dos caracteres, estes devem poder ser lidos e escritos
 livremente.
Essa funcionalidade se resume, basicamente, e dois tipos de objetos: um que lê
 caracteres de uma fonte, e outro que escreve caracteres para uma fonte.
Tais objetos são chamados [[streams]].
```

\subsection{Buffers}

De acordo com a estrutura abstrata descrita acima, [[buffers]] são
 estruturas simples, cuja única funcionalidade é armazenar quantidades
 indefinidas de texto.
Um [[buffer]] deve ser composto, então, de uma área de memória para
 armazenagem, um indicador de seu tamanho, um indicador de sua ocupação e uma
 referência para um outro [[buffer]] que possa continuar a armazenar o
 texto caso sua memória se esgote.

Os únicos métodos necessários em um [[buffer]] são o construtor, que
 deverá inicializar os atributos do objeto, e o destrutor, que deve liberar
 os demais [[buffers]] na lista.
Nenhum outro será necessário pois o [[buffer]] é um elemento passivo,
 sendo manipulado pelo [[streams]] exclusivamente.
Dessa forma, os dados internos dos [[buffers]] devem ter sua
 visibilidade restrita aos [[streams]].

```
<<Classe buffer>>=
class buffer
{
  <<Descrição de buffer>>
}; @

<<Descrição de buffer>>=
private:
  buffer *next;
  int size;
  int pos;
  char data[BUFFER_SIZE];
public:
  buffer()
  {
    next = NULL;
    size = BUFFER_SIZE;
    pos = 0;
  }
  ~buffer()
  {
    if (next) delete(next);
  } @
```

A esta definição básica de um [[buffer]], podemos adicionar um
 construtor que receba uma string e inicialize o objeto para contê-la.
Este método simplifica bastante a criação de [[buffers]] para armazenar
 strings pré-definidas.

```
<<Descrição de buffer>>=
public:
  buffer(char *s)
  {
    size = BUFFER_SIZE;
    int len = strlen(s);
    if (len <= BUFFER_SIZE)
    {
      memcpy(data, s, len);
      next = NULL;
      pos = len;
    }
    else
    {
      memcpy(data, s, BUFFER_SIZE);
      next = new buffer(s+BUFFER_SIZE);
      pos = BUFFER_SIZE;
    }
  } @

Uma característica que se deve ter em mente é que tais objetos serão usados,
 na maior parte das vezes, para armazenar pequenas quantidades de texto.
Isso porque, tipicamente, estaremos armazenando nomes de macros e seus
 parâmetros.
Assim, vamos usar um tamanho de [[buffer]] pequeno e compatível com esta
 característica.

<<Tamanho da área de dados dos buffers>>=
#define BUFFER_SIZE 20 @
```

A.2 Transformação Efetuada por `code`

O resultado do processamento do fonte exibido na seção anterior, pelo programa `code`, leva ao texto que se segue, apresentado em fonte de máquina.

```
#line 94 "text.w"
#define BUFFER_SIZE 20
#line 487 "text.w"


#line 38 "text.w"
class buffer
{

#line 44 "text.w"
private:
  buffer *next;
```

```
  int size;
  int pos;
  char data[BUFFER_SIZE];
public:
  buffer()
  {
    next = NULL;
    size = BUFFER_SIZE;
    pos = 0;
  }
  ~buffer()
  {
    if (next) delete(next);
  }
#line 67 "text.w"
public:
  buffer(char *s)
  {
    size = BUFFER_SIZE;
    int len = strlen(s);
    if (len <= BUFFER_SIZE)
    {
      memcpy(data, s, len);
      next = NULL;
      pos = len;
    }
    else
    {
      memcpy(data, s, BUFFER_SIZE);
      next = new buffer(s+BUFFER_SIZE);
      pos = BUFFER_SIZE;
    }
  }
#line 233 "text.w"
  friend class bufferReadStream;
#line 424 "text.w"
  friend class bufferWriteStream;
#line 40 "text.w"

};
#line 488 "text.w"
```

A.3 Transformação Efetuada por `doc`

A seguir é apresentado, em fonte de máquina, o resultado do processamento do fonte pelo programa `doc`.

A gerência do texto pode ser dividida em dois problemas principais a serem resolvidos: a armazenagem e a manipulação.
Quando o texto tem sua origem, digamos, em arquivos, não precisamos nos preocupar com armazenagem mas apenas com sua manipulação.
A mesma situação se repete quando se trata de texto a ser enviado para um arquivo, ou até mesmo quando se trata de algum outro elemento de armazenagem.
Porém, quando este não for o caso, devemos disponibilizar ao usuário uma forma de armazenagem.
Isso pode ser conseguido através de objetos chamados \verb|buffers|.
Tais objetos têm, como único objetivo, armazenar texto para manipulação.

Já o processo de manipulação deve ser todo criado pois, não importando a fonte (ou o destino) dos caracteres, estes devem poder ser lidos e escritos livremente.
Essa funcionalidade se resume, basicamente, e dois tipos de objetos: um que lê caracteres de uma fonte, e outro que escreve caracteres para uma fonte.
Tais objetos são chamados \verb|streams|.

\subsection{Buffers}

De acordo com a estrutura abstrata descrita acima, \verb|buffers| são estruturas simples, cuja única funcionalidade é armazenar quantidades indefinidas de texto.
Um \verb|buffer| deve ser composto, então, de uma área de memória para armazenagem, um indicador de seu tamanho, um indicador de sua ocupação e uma referência para um outro \verb|buffer| que possa continuar a armazenar o texto caso sua memória se esgote.

Os únicos métodos necessários em um \verb|buffer| são o construtor, que deverá inicializar os atributos do objeto, e o destrutor, que deve liberar os demais \verb|buffers| na lista.
Nenhum outro será necessário pois o \verb|buffer| é um elemento passivo, sendo manipulado pelo \verb|streams| exclusivamente.
Dessa forma, os dados internos dos \verb|buffers| devem ter sua visibilidade restrita aos \verb|streams|.

```
\begin{flushleft}\noindent$\langle${\it Classe buffer
\refstepcounter{docCounter}\label{@@doc:text.wa:1}\thedocCounter}
$\rangle\equiv$\\
\verb|  class buffer|\\
\verb|  {|\\
\verb|    |$\langle${\it Descrição de buffer \ref{@@doc:text.wa:2}}$\rangle$
\verb||\\
\verb|  }; |\\{\small Este trecho é usado em \ref{@@doc:text.wa:29}.}
\end{flushleft}
```

```
\begin{flushleft}\noindent$\langle${\it Descrição de buffer
\refstepcounter{docCounter}
\label{@@doc:text.wa:2}\thedocCounter}$\rangle\equiv$\\
\verb|  private:|\\
\verb|    buffer *next;|\\
\verb|    int size;|\\
\verb|    int pos;|\\
\verb|    char data[BUFFER_SIZE];|\\
\verb|  public:|\\
\verb|    buffer()|\\
\verb|    {|\\
\verb|      next = NULL;|\\
\verb|      size = BUFFER_SIZE;|\\
\verb|      pos = 0;|\\
\verb|    }|\\
\verb|    ~buffer()|\\
\verb|    {|\\
\verb|      if (next) delete(next);|\\
\verb|    } |\\{\small Este trecho é usado em \ref{@@doc:text.wa:1}\\
Este trecho é continuado em \ref{@@doc:text.wa:3} \ref{@@doc:text.wa:13}
\ref{@@doc:text.wa:25}.}\end{flushleft}

A esta definição básica de um \verb|buffer|, podemos adicionar um
 construtor que receba uma string e inicialize o objeto para contê-la.
Este método simplifica bastante a criação de \verb|buffers| para armazenar
 strings pré-definidas.



\begin{flushleft}\noindent$\langle${\it Descrição de buffer
\refstepcounter{docCounter}
\label{@@doc:text.wa:3}\thedocCounter}$\rangle+\equiv$\\
\verb|  public:|\\
\verb|    buffer(char *s)|\\
\verb|    {|\\
\verb|      size = BUFFER_SIZE;|\\
\verb|      int len = strlen(s);|\\
\verb|      if (len <= BUFFER_SIZE)|\\
\verb|      {|\\
\verb|        memcpy(data, s, len);|\\
\verb|        next = NULL;|\\
\verb|        pos = len;|\\
\verb|      }|\\
\verb|      else|\\
```

```
\verb|      {|\\
\verb|        memcpy(data, s, BUFFER_SIZE);|\\
\verb|        next = new buffer(s+BUFFER_SIZE);|\\
\verb|        pos = BUFFER_SIZE;|\\
\verb|      }|\\
\verb|    } |\\{\small Este trecho é continuação de \ref{@@doc:text.wa:2}.}
\end{flushleft}

Uma característica que se deve ter em mente é que tais objetos serão usados,
 na maior parte das vezes, para armazenar pequenas quantidades de texto.
Isso porque, tipicamente, estaremos armazenando nomes de macros e seus
 parâmetros.
Assim, vamos usar um tamanho de \verb|buffer| pequeno e compatível com esta
 característica.



\begin{flushleft}\noindent$\langle${\it Tamanho da área de dados dos buffers
\refstepcounter{docCounter}\label{@@doc:text.wa:4}\thedocCounter}$\rangle
\equiv$\\
\verb|  #define BUFFER_SIZE 20 |\\{\small Este trecho é usado em
\ref{@@doc:text.wa:29}.}\end{flushleft}
```

A.4 Documentação Final Gerada por LaTeX

Todo o conteúdo desta seção, a começar após a linha abaixo, é resultado do processamento pelo LaTeXdo fonte gerado por `doc`.

A gerência do texto pode ser dividida em dois problemas principais a serem resolvidos: a armazenagem e a manipulação. Quando o texto tem sua origem, digamos, em arquivos, não precisamos nos preocupar com armazenagem mas apenas com sua manipulação. A mesma situação se repete quando se trata de texto a ser enviado para um arquivo, ou até mesmo quando se trata de algum outro elemento de armazenagem. Porém, quando este não for o caso, devemos disponibilizar ao usuário uma forma de armazenagem. Isso pode ser conseguido através de objetos chamados `buffers`. Tais objetos têm, como único objetivo, armazenar texto para manipulação.

Já o processo de manipulação deve ser todo criado pois, não importando a fonte (ou o destino) dos caracteres, estes devem poder ser lidos e escritos livremente. Essa funcionalidade se resume, basicamente, e dois tipos de objetos: um que lê caracteres de uma fonte, e outro que escreve caracteres para uma fonte. Tais objetos são chamados `streams`.

Buffers

De acordo com a estrutura abstrata descrita acima, `buffers` são estruturas simples, cuja única funcionalidade é armazenar quantidades indefinidas de texto. Um `buffer` deve ser composto, então, de uma área de memória para armazenagem, um indicador de seu tamanho, um indicador de sua ocupação e uma referência para um outro `buffer` que possa continuar a armazenar o texto caso sua memória se esgote.

Os únicos métodos necessários em um `buffer` são o construtor, que deverá inicializar os atributos do objeto, e o destrutor, que deve liberar os demais `buffers` na lista. Nenhum outro será necessário pois o `buffer` é um elemento passivo, sendo manipulado pelo `streams` exclusivamente. Dessa forma, os dados internos dos `buffers` devem ter sua visibilidade restrita aos `streams`.

⟨*Classe buffer 45a*⟩ ≡

```
class buffer
{
  ⟨Descrição de buffer 45b⟩
};
```

Este trecho é usado em ??.

⟨*Descrição de buffer 45b*⟩ ≡

```
private:
  buffer *next;
  int size;
  int pos;
  char data[BUFFER_SIZE];
public:
  buffer()
  {
    next = NULL;
    size = BUFFER_SIZE;
    pos = 0;
  }
  ~buffer()
  {
    if (next) delete(next);
  }
```

Este trecho é usado em 45a

Este trecho é continuado em 45c ?? ??.

A esta definição básica de um `buffer`, podemos adicionar um construtor que receba uma string e inicialize o objeto para contê-la. Este método simplifica bastante a criação de `buffers` para armazenar strings pré-definidas.

⟨*Descrição de buffer 45c*⟩+ ≡

```
public:
  buffer(char *s)
  {
    size = BUFFER_SIZE;
    int len = strlen(s);
    if (len <= BUFFER_SIZE)
    {
      memcpy(data, s, len);
```

```
      next = NULL;
      pos = len;
    }
    else
    {
      memcpy(data, s, BUFFER_SIZE);
      next = new buffer(s+BUFFER_SIZE);
      pos = BUFFER_SIZE;
    }
  }
```
Este trecho é continuação de 45b.

Uma característica que se deve ter em mente é que tais objetos serão usados, na maior parte das vezes, para armazenar pequenas quantidades de texto. Isso porque, tipicamente, estaremos armazenando nomes de macros e seus parâmetros. Assim, vamos usar um tamanho de `buffer` pequeno e compatível com esta característica.

⟨*Tamanho da área de dados dos buffers 46a*⟩ ≡
```
#define BUFFER_SIZE 20
```
Este trecho é usado em **??**.

Apêndice B

Exemplos de Utilização

São apresentados nesse apêndice exemplos reais de utilização da técnica de programação literária que utilizaram a ferramenta **nome** como suporte. Inicialmente são mostrados, na seção B.1, trechos da implementação do framework de suporte à construção de ferramentas de programação literária apresentado neste trabalho. As demais seções apresentam passagens de outros trabalhos desenvolvidos no TeCGraf.

A seção B.2 introduz o pacote IUP/Lua [Gor95], desenvolvido por Tomás Guisasola Gorham. Também desenvolvido por este autor, é apresentado na seção B.3 o sistema LDB [Gor96b]. Na seção B.4 é apresentado um *toolkit* para criação de interfaces gráficas com o usuário [dC96], desenvolvido por André Oliveira da Costa.

Deve-se notar que uso do sistema de referências cruzadas do LaTeX faz com que a identificação dos trechos impressos esteja correta, uma vez que são formados pelo número da página seguido por um caracter diferenciador. Outro ponto a ser notado é que diversas referências não puderam ser concluídas, visto que grande parte do código está ausente, o que é indicado através de duplos pontos de interrogação no local onde deveria haver uma referência.

Todo material de que se compõem as próximas seções deste capítulo são partes dos programas citados. Os títulos das seções dão dicas sobre o que trata o conteúdo. Esses códigos retirados dos projetos citados se encontram delimitados por linhas horizontais, como no exemplo abaixo. As linhas delimitadoras foram introduzidas apenas para aumentar a clareza.

Esse código foi retirado do projeto XX.

B.1 Framework de Suporte à Programação Literária

B.1.1 Trecho da definição de streams

A idéia por trás dos `streams` é permitirmos que usuário leia ou escreva caracteres de uma forma homogênea, independente da origem ou do destino que estes caracteres venham a ter. Dessa forma, apenas dois tipos básicos devem criados: um que representa a classe de objetos de lêem caracteres, e outro os que os escrevem.

⟨*Classe readStream 48a*⟩ ≡

```
class readStream
{
  ⟨Descrição de readStream 48c⟩
};
```

Este trecho é usado em **??**.

⟨*Classe writeStream 48b*⟩ ≡

```
class writeStream
{
  ⟨Descrição de writeStream ??⟩
};
```

Este trecho é usado em **??**.

Streams de Leitura

A classe de `stream` de leitura poderia, a princípio, ser implementada com um único método virtual que retornasse um caracter, sendo esse método especializado em cada subclasse de acordo com a fonte dos caracteres. Porém, tal implementação implicaria em uma chamada de função a cada caracter a ser lido: uma impossibilidade em termos de performance. Uma estrutura muito mais eficaz pode ser obtida com o uso de uma área de memória e um contador do número de caracteres armazenados. Basta usarmos essa área como um local de armazenamento temporário e prover um método inline que retorne os caracteres dali. Dessa forma, a leitura de um caracter de um `stream` tem, na maior parte das vezes, o mesmo custo de um acesso a um vetor. Obviamente, uma chamada a um método virtual é necessária para encher a área de memória mas isso só ocorrerá um número de vezes irrisório frente ao número de caracteres lidos.

Para implementarmos essa política, a classe de `stream` de leitura deve possuir um ponteiro para os caracteres a serem lidos, o número de caracteres a ler e o método que será chamado para preencher a área de memória do `stream` com os caracteres lidos da fonte. Esses dados devem ser de uso exclusivo dos objetos da classe, através da função de leitura de caracter que, por sua vez, deverá ser codificada em cada uma das especializações da classe.

Note que *não há* uma área de memória no `stream`. Isso acontece porque, dependendo da fonte dos caracteres, uma especialização da classe pode usar uma área de dados da própria fonte. Dessa forma, fica a critério das especializações criar ou não uma área própria de memória para o armazenamento temporário.

⟨*Descrição de readStream 48c*⟩ ≡

```
protected:
  char *endData;
  int pos;
  virtual int read() = 0;
```

Este trecho é usado em 48a
Este trecho é continuado em 49a **??**.

A estrutura de ponteiro para memória e posição foi pensada para permitir um acesso rápido aos caracteres — uma vez que performance é uma das diretrizes básicas do sistema. Isso porque, com um ponteiro para um final do texto e mantendo o inteiro do número de caracteres a ler com valor negativo, podemos verificar se há mais caracteres a ler, indexar direto o vetor ou chamar a função de leitura, tudo de uma forma clara e eficaz.

⟨*Descrição de readStream 49a*⟩+ ≡
```
public:
  inline int get_char()
  {
    return (pos ? endData[++pos] : read());
  }
```
Este trecho é continuação de 48c.

B.1.2 Trecho da definição de ambientes

As estruturas de definição de ambiente devem atender o critério básico de poder determinar rapidamente se um caracter lido na entrada começa um `mark`. Como uma política muito razoável de construção desses nomes é usar um caracter constante no início de todos eles, podemos criar, na base da estrutura, um vetor de ponteiros para árvores de nomes. Desta forma, para sabermos se um dado caracter inicia um `mark`, precisamos apenas indexar este vetor.

A árvore deve, então, guiar a busca caracter a caracter até chegarmos ao nome completo. Segundo essa política, podemos colocar em cada nó da árvore informações sobre o caracter corrente, a ação a ser executada (caso um `mark` tenha se completado ali), seus filhos, seus irmãos e sua altura.

O caracter guia a busca, a ação permite disparar o `mark`, a lista de filhos permite a continuação da busca quando um novo caracter for lido, a lista de irmãos possibilita que vários `marks` se completem e, finalmente, a altura permite aplicar os critérios de desempate entre dois `marks` identificados simultaneamente na entrada.

Dessa forma, dispersamos ao longo da árvore, de uma forma eficiente, todas as informações relativas a um `mark`. Podemos, então, incorporar nos ambientes a funcionalidade dos `marks`, dispensando o uso desta classe em separado.

Em termos de funcionalidade, um objeto `markSet` deve possuir um construtor que inicialize seus dados, além de um método que registre um `mark`. Este método de registro de `marks` pode ser implementado para receber tanto uma string quanto um `buffer` que contém seu nome. Com as informações expostas, podemos definir a estrutura do ambiente.

⟨*Classe markSet 49b*⟩ ≡
```
class markSet
{
  ⟨Descrição de markSet 50a⟩
};
```
Este trecho é usado em **??**.

⟨*Descrição de markSet 50a*⟩ ≡

```
private:
  treeNode *base[256];
public:
  markSet()
  {
    for (int i=0; i<256; base[i++]=NULL);
  }
  void insert_mark(char *mark, action *f);
  void insert_mark(buffer *mark, action *f);
```

Este trecho é usado em 49b
Este trecho é continuado em **??** 52a.

B.1.3 Trecho da definição de pilhas

Definidos os ambientes, devemos criar uma estrutura de pilha e especificar uma forma de busca de `marks` através desta pilha. Como a funcionalidade de empilhar um ambiente sugere que esta será uma operação feita raramente em muita profundidade, podemos limitar a profundidade máxima da pilha criando-a como um vetor de referências para ambientes.

⟨*Profundidade máxima da pilha 50b*⟩ ≡

```
#define STACK_SIZE 256
```

Este trecho é usado em **??**.

Com relação a sua funcionalidade, a pilha é uma classe simples, só dispondo de métodos para especificar os `streams` de entrada e saída, empilhar e desempilhar ambientes, além de um método responsável por processar o `stream` de entrada, passando o resultado para o `stream` de saída. Para facilitar a programação, podemos fazer com que os métodos responsáveis pela alteração dos `streams` retornem os anteriores.

⟨*Classe setStack 50c*⟩ ≡

```
class setStack
{
  ⟨Descrição de setStack 50d⟩
};
```

Este trecho é usado em **??**.

⟨*Descrição de setStack 50d*⟩ ≡

```
private:
  markSet *stack[STACK_SIZE];
  int stack_pos;
  readStream *input;
  writeStream *output;
public:
```

```
readStream *set_input(readStream *in)
{
  readStream *old = input;
  input = in;
  return old;
}
writeStream *set_output(writeStream *out)
{
  writeStream *old = output;
  output = out;
  return old;
}
void push_mark_set(markSet *ms);
markSet *pop_mark_set();
void process();
setStack()
{
  stack_pos = 0;
  input = NULL;
  output = NULL;
}
```
Este trecho é usado em 50c
Este trecho é continuado em ??.

B.1.4 Trecho da implementação da máquina de estados

Definimos, agora, o método de processamento `process`, responsável pelo processamento do texto de entrada, transferindo o resultado para a saída. Este método deve, a cada caracter lido, verificar se os caminhos em andamento alcançam um estado final em alguma das árvores de busca, executando, neste caso, a ação associada ao `mark`. Caso nenhum `mark` seja alcançado, a função deve retirar do vetor de busca os caminhos que não podem mais ser percorridos, além de acrescentar todos os que puderem começar no caracter em questão (passo responsável pelo back-tracking). Além disso, sempre que descobrirmos ser impossível alcançar um `mark` a partir da entrada, devemos passar os caracteres lidos diretamente para a saída.

Note que a classe que implementa a pilha de ambientes deve ser feita `friend` das classes `treeNode` e `macroSet` para poder acessar os dados internos dessas estruturas. Isso é necessário pois, caso esse acesso fosse feito via métodos a performance seria sensivelmente pior.

⟨*Descrição de treeNode 51a*⟩+ ≡
```
friend class setStack;
```
Este trecho é continuação de ??.

⟨*Descrição de markSet 52a*⟩+ ≡

```
    friend class setStack;
```

Este trecho é continuação de 50a.

Para armazenar os caracteres lidos durante a busca de um `mark` optamos por um array local. Essa abordagem se justifica uma vez que toda a complexidade de gerência de `buffers` será posta de lado — aumentando significativamente a performance do loop principal. Note que dessa forma impomos um limite máximo para o comprimento de um `mark`.

⟨*Comprimento máximo de um mark 52b*⟩ ≡

```
  #define MARK_SIZE 256
```

Este trecho é usado em **??**.

⟨*Implementação de setStack::process 52c*⟩ ≡

```
  void setStack::process()
  {
    char buf[MARK_SIZE+1]; // +1 para o '\0' terminador
    int buf_pos = 0;
    int c;
    Loop_principal:
    while ((c = input->get_char()) != EOF)
    {
      ⟨Expande os caminhos existentes em busca de um mark e o executa 52d⟩
      ⟨Retira os caminhos que não podem mais ser percorridos ??⟩
      ⟨Procura novos caminhos começando no caracter lido ??⟩
      ⟨Se não há caminhos a percorrer, copia os caracteres lidos para a saída ??⟩
    }
    ⟨Copia os caracteres lidos para a saída ??⟩
  }
```

Este trecho é usado em **??**.

Para expandir os caminhos devemos, em cada um dos caminhos sendo analisados pela máquina, percorrer a lista corrente em busca do caracter lido. Caso seja encontrado, devemos verificar se há uma ação cadastrada pois isso indicará que a seqüência lida até o caracter em questão é um `mark`. Nesse caso, a ação deve ser executada, a máquina reinicializada e o controle deve voltar ao início do loop — pois não haverá caminhos a serem retirados (a máquina estará vazia) e não desejamos criar novos caminhos a partir do caracter corrente (pois ele já terá sido utilizado). Se o caracter não for encontrado, devemos atribuir um valor nulo à entrada correspondente da máquina, indicando que o caminho não pode mais ser percorrido.

⟨*Expande os caminhos existentes em busca de um* `mark` *e o executa 52d*⟩ ≡

```
  {
    for (int q=0; q<search_pos; q++)
    {
      search[q].tree = search[q].tree->child;
      if (search[q].tree)
```

```
      {
        while (search[q].tree && search[q].tree->c != c)
          search[q].tree = search[q].tree->brother;
        if (search[q].tree)
        {
          if (search[q].tree->f)
          {
            for (int i=0; i<search[q].firstChar; i++)
              output->put_char(buf[i]);
            buf[buf_pos++] = c;
            buf[buf_pos++] = '\0';
            search[q].tree->f->do_it(buf+search[q].firstChar);
            ⟨Reinicializa a busca de um mark 53a⟩
          }
        }
      }
    }
  }
```
Este trecho é usado em 52c.

Onde reinicializar a máquina de busca é: fazer seu tamanho igual à zero, esvaziar a área auxiliar que contém os caracteres lidos e voltar a execução para o início do loop.

⟨*Reinicializa a busca de um* `mark` *53a*⟩ ≡
```
  {
    search_pos = 0;
    buf_pos = 0;
    goto Loop_principal;
  }
```
Este trecho é usado em 52d **??**.

B.2 IUP/Lua

B.2.1 Resumo

Este trabalho apresenta uma interface entre a linguagem de programação Lua e o sistema portátil de interface com o usuário, IUP. O principal objetivo deste pacote é fornecer facilidades para a construção de diálogos IUP usando a linguagem Lua. A idéia de se usar Lua como linguagem de configuração, partiu das limitações encontradas na linguagem LED para a definição de ações para tratamento dos eventos.

Foram usadas abstrações que tentam criar um ambiente de programação semelhante ao de linguagens orientadas a objetos. O conceito de programação orientada a eventos é muito usado, devido ao fato da biblioteca IUP estar baseada neste modelo. Apesar deste sistema vir a ocupar o espaço atualmente ocupado pela linguagem LED, a maior parte das construções usadas em Lua foram fortemente baseadas nas construções correspondentes em LED.

B.2.2 Trecho da definição da arquitetura

Os tipos de elementos de interface IUP são representados como classes em Lua, logo, a criação de um elemento de interface equivale à criação de um objeto de uma das classes previamente definidas. A hierarquia de classes definida no sistema tem como raiz a classe `WIDGET`. Essa classe define algumas funções e atributos comuns a todas as classes. O atributo `parent` é usado para implementar o mecanismo de herança em Lua (veja mais detalhes no apêndice **??**). O atributo `handle` guarda o *handle* do objeto IUP associado. O atributo `type` é usado para fazer a checagem dos parâmetros, como será detalhado mais adiante na seção B.2.3.

⟨*WIDGET Class definition 54a*⟩ ≡
```
WIDGET = {type = {}}
```
  ⟨*WIDGET Constructor 54c*⟩
  ⟨*WIDGET parameter checking 55b*⟩
  ⟨*WIDGET set extra attributes* **??**⟩
  ⟨*WIDGET methods* **??**⟩

Este trecho é usado em **??**.

Como subclasses de `WIDGET`, temos, por exemplo, as classes `IUPBUTTON`, `IUPDIALOG` e `IUPCANVAS`. Os objetos dessas subclasses são criados a partir de chamadas em Lua semelhantes às construções em LED, como no exemplo a seguir:

⟨*first example 54b*⟩ ≡
```
b = iupbutton{title = "Ok", fgcolor = "255 0 0"}
d = iupdialog{b}
```
Este trecho é usado em **??**
Este trecho é continuado em **?? ??**.

B.2.3 Trecho da definição de elementos genéricos

De uma maneira geral, os elementos de interface serão criados através de uma chamada ao construtor da classe `WIDGET`. Este método pode ser dividido em seis partes básicas: estabelecer a herança (mais detalhes no apêndice **??**), checar parâmetros, criar o elemento IUP, definir os atributos extra, guardar uma referência do objeto e retornar o handle.

⟨*WIDGET Constructor 54c*⟩ ≡
```
function WIDGET:Constructor (obj)
  obj.parent = self
```
    ⟨*check Widget parameters 55a*⟩

```
    ⟨create IUP Widget ??⟩
    ⟨set Widget extra attributes ??⟩
    ⟨saving reference ??⟩
    return obj.handle
  end
```
Este trecho é usado em 54a.

Vale ressaltar que o parâmetro `obj`, no método acima, é o objeto que está sendo criado e a variável `self` é a *classe* que está criando o elemento.

Checando parâmetros

O segundo passo é fazer a checagem da presença dos parâmetros (atributos obrigatórios) e seus tipos. Essa checagem é fundamental, já que esses valores serão passados para alguma rotina em C que efetivamente criará o elemento.

⟨*check Widget parameters 55a*⟩ ≡
```
  self:checkParams (obj)
```
Este trecho é usado em 54c.

A função de verificação de parâmetros da classe `WIDGET` pode ser usada para todos os elementos de interface. Para isto, uma estrutura (guardada no campo `type`) da classe possui um campo com o nome de cada parâmetro do elemento de interface. Esses campos guardam funções que checam os tipos dos campos correspondentes do objeto. A função genérica para a checagem dos parâmetros é definida como:

⟨*WIDGET parameter checking 55b*⟩ ≡
```
  function WIDGET:checkParams (obj)
    local type = self.type
    local param, func = next(type, nil)
    while param do
      if not func(obj[param]) then
        error("parameter " .. param .. " has wrong value or is not initialized")
      end
      param, func = next(type, param)
    end
  end
```
Este trecho é usado em 54a.

Cada campo contido na tabela `type` da classe guarda uma função que é usada para testar o campo de mesmo nome do objeto ("instância da classe").

Para entender melhor o funcionamento do método de checagem de parâmetros, veja a declaração de um botão

```
  b = iupbutton{title = "Ok", fgcolor = RED}
```

e a definição da classe `IUPBUTTON`:

⟨*primitive classes 56a*⟩ ≡

```
IUPBUTTON = {parent = WIDGET,
       type = {title = type_string}}
```

⟨*BUTTON CreateIUPelement* ??⟩
⟨*BUTTON methods* ??⟩
⟨*BUTTON constructor* ??⟩
⟨*BUTTON default action* ??⟩

Este trecho é usado em ??
Este trecho é continuado em ?? ?? ?? ?? ?? ?? ?? ?? ?? ??.

Essa declaração mostra que um botão é uma subclasse de `WIDGET` (`parent = WIDGET`) e que é obrigatória a presença do atributo `title` (do tipo `type_string`).

O método `checkParams`, herdado de `WIDGET` percorre todos os campos contidos na estrutura `type` da classe do elemento (no caso, `IUPBUTTON`), executando a função contida no mesmo. Por exemplo, ao identificar o campo `title` na classe `IUPBUTTON`, a função `type_string` será chamada, recebendo como argumento o valor do campo `title` do objeto `b`, que vale "Ok". A função `type_string` é definida como a seguir:

⟨*type checking functions 56b*⟩ ≡

```
function type_string (o) return type(o) == "string" end
```

Este trecho é usado em ??
Este trecho é continuado em ?? ?? ??.

B.3 LDB

B.3.1 Trecho da introdução

Este trabalho apresenta toda a implementação do depurador para a linguagem de extensão Lua. O depurador foi construído com base na interface de depuração definida na versão 2.3 da linguagem [IdFF95]. Ele é oferecido como uma biblioteca Lua onde todos os recursos de depuração são funções da linguagem.

A linguagem de comandos do depurador é a própria linguagem Lua, o que facilita a manipulação das variáveis e estruturas de dados. Além disso, é possível avaliar expressões, executar funções ou até trechos de código definidos pelo usuário.

Este trabalho é a implementação do sistema desenvolvido para a dissertação de mestrado de seu autor [Gor96a].

B.3.2 Trecho da definição da arquitetura

Um depurador tem três funções principais: controlar a execução do código (permitindo execução passo-a-passo), mostrar o ponto do código fonte que será executado e acessar as variáveis locais e globais. As outras funções, como por exemplo, pontos de parada condicionais, inspetores de estruturas de dados e visualização de variáveis locais a outras funções, podem ser todas derivadas dessas fundamentais.

O interpretador Lua oferece alguns meios de se implementar essas funcionalidades. Para controlar a execução do código, o depurador utiliza o mecanismo de ganchos[1], que permite que, em determinados pontos do código, a execução seja transferida para a aplicação através da chamada de uma função. Instruções que indiquem as mudanças de linha no código fonte podem ser geradas através do uso da diretiva de compilação `$debug`. Alia-se a isso a facilidade de se ter o interpretador Lua disponível para executar comandos da linguagem, para acessar e alterar o conteúdo das variáveis do programa, chamar funções etc.

Para entender melhor o funcionamento do depurador, são definidos alguns conceitos importantes: ganchos, pontos de código, ações e ambiente de execução.

Os ganchos

O depurador de Lua utiliza dois ganchos oferecidos pela interface de depuração da linguagem:

Mudança de linha de execução
Chamada e retorno de função

Os ganchos são os únicos pontos onde a execução pode ser interrompida pelo depurador. O depurador define duas funções que são os *tratadores* dos ganchos. Estas funções gerenciam a execução do código, podendo interrompê-la e permitir a interação do usuário.

Pontos de código

Pontos de código são pontos do código fonte de um programa. O depurador utiliza pontos de código para informar o usuário qual ponto do programa está sendo depurado. Um ponto de código também é usado por vários comandos de depuração que precisam referenciar algum ponto do código fonte, tais como `break`, `goto` e `list`.

Um ponto de código é normalmente indicado por uma linha e um nome de arquivo, mas ele também pode ser identificado por uma função. Neste caso, a função é convertida pelo depurador em um par ⟨linha, arquivo⟩ que indica onde a função foi definida.

O depurador possui o conceito de um ponto de código corrente, que indica o ponto que está sendo depurado. Assim sendo, todos os comandos que precisam de um ponto de código aceitam as três formas a seguir:

⟨linha, arquivo⟩ que indica a linha e arquivo dados.

⟨linha⟩ que indica a linha dada do arquivo corrente.

⟨função⟩ que indica a linha e o arquivo onde a função foi definida.

Existem algumas situações especiais onde o depurador indica uma localização não óbvia. Um destes casos é no retorno de uma função, quando o depurador usa a string `<return>` como arquivo corrente. Outro caso é na execução de um arquivo pela função `dofile`, quando a linha corrente passa a valer zero. No caso da execução de uma string pela função `dostring`, o depurador define o arquivo corrente como sendo a string `(string)`.

A estrutura `location` representa um ponto de código.

[1] Mais detalhes sobre o funcionamento deste mecanismo podem ser encontrados em [Gor96a]

⟨*data structures 57a*⟩ ≡
```
typedef struct {
  int line;
  char *file;
} location;
```
Este trecho é usado em ??
Este trecho é continuado em ?? ??.

A variável `curr_loc` indica o ponto de execução do código Lua.

⟨*global variables 58a*⟩ ≡
```
static location curr_loc = {0, NULL};
```
Este trecho é usado em ??
Este trecho é continuado em 58c 59a 59b 59d 59e ?? ?? ?? ?? ?? ??.

A função `update_location` atualiza a posição do depurador de acordo com o nível e a linha dados. Para fazer esta atualização, chama-se a função `lua_funcinfo` sobre a função no nível dado. Se a linha não for conhecida (`line` == -1), tentamos obtê-la através de `lua_currentline`.

⟨*support functions 58b*⟩ ≡
```
static void update_location (int level, int line)
{
  lua_Object f = lua_stackedfunction (level);
  lua_funcinfo (f, &curr_loc.file, &curr_loc.line);
  if ((line != -1)  || ((line = lua_currentline (f)) != -1))
    curr_loc.line = line;
}
```
Este trecho é usado em ??
Este trecho é continuado em ??.

B.3.3 Trecho da definição de ganchos

Os comandos de depuração são apenas funções Lua que sinalizam para os tratadores o que eles devem fazer. Esta sinalização foi viabilizada através do uso de algumas variáveis globais que compõem o estado do depurador.

A variável `stop_now` é um flag de parada para qualquer gancho do depurador.

⟨*global variables 58c*⟩+ ≡
```
static int stop_now = 0;
```
Este trecho é continuação de 58a.

A variável `line_count` é usada para contar linhas que devem ser executadas sem interrupção, como no caso dos comandos `next` e `step` com parâmetros.

⟨*global variables 59a*⟩+ ≡
```
static int line_count;
```
Este trecho é continuação de 58a.

A variável `mode` indica qual o modo de execução está sendo usado no momento pelo depurador. Isto é importante para fazer o controle adequado dos contadores de linhas e de níveis de funções.

⟨*global variables 59b*⟩+ ≡
```
enum {RUN_MODE, STEP_MODE, NEXT_MODE} mode = RUN_MODE;
```
Este trecho é continuação de 58a.

Para facilitar o uso, são definidas as macros:

⟨*definitions 59c*⟩ ≡
```
#define IN_RUN_MODE (mode == RUN_MODE)
#define IN_STEP_MODE (mode == STEP_MODE)
#define IN_NEXT_MODE (mode == NEXT_MODE)
```
Este trecho é usado em **??**
Este trecho é continuado em **?? ?? ?? ?? ?? ??**.

As variáveis `line_on` e `call_on` indicam se os respectivos ganchos estão ligados ou não. Elas são usadas para evitar que alguma função ative ou desative os ganchos durante a execução de um deles, o que pode causar a chamada recursiva destes. Elas podem assumir um estado que indica ao tratador que desligou os ganchos de que eles devem permanecer desligados (`DONT_TURN_ON`).

⟨*global variables 59d*⟩+ ≡
```
enum {OFF_HOOK, ON_HOOK, DONT_TURN_ON} line_on = OFF_HOOK,
                                       call_on = OFF_HOOK;
```
Este trecho é continuação de 58a.

A variável `func_level` é usada para contar quantas funções estão na pilha acima da função em que foi dado o comando `next`.

⟨*global variables 59e*⟩+ ≡
```
static int func_level;
```
Este trecho é continuação de 58a.

B.4 Toolkit para Construção de Diálogos

B.4.1 Trecho da introdução

Este trabalho se propõe a implementar um *toolkit* de pequeno porte para a criação de interfaces gráficas com o usuário (ou GUIs, de *Graphical User Interfaces*). Este *toolkit* deve seguir o modelo de *boxes and glue* implementado em TEX [Knu86] — mais especificamente, o modelo a ser seguido é o de IUP [LdFG+96][2]

Quando dizemos que será um "*toolkit* de pequeno porte", o fazemos porque implementaremos um subconjunto bastante reduzido dos *widgets* oferecidos por IUP. Por se tratar de um *toolkit* já bastante "maduro" (trata-se de um produto utilizado na construção da interface da grande maioria dos sistemas desenvolvidos pelo TeCGraf), IUP oferece uma gama bastante variada de *widgets*: *push-buttons*, *toggles*, *text-boxes*, *list-boxes*, *canvas* etc. Nosso foco no desenvolvimento do *toolkit* que é tema deste trabalho não será a disponibilização de uma gama variada de *widgets*, mas sim o algoritmo de posicionamento dos mesmos na interface.

O modelo seguido por IUP para descrever o *layout* dos elementos se baseia nas abstrações de `hboxes`, `vboxes` e `fills`:

hboxes: "caixas" onde os elementos nela contidos serão exibidos lado a lado

vboxes: "caixas" onde os elementos nela contidos serão exibidos segundo uma orientação vertical

fills: "molas" que têm como finalidade propiciar um mecanismo fácil e intuitivo para definir o espaçamento entre os objetos de interface

---

B.4.2 Trecho da implementação de orientação a objetos em Lua

Lua não é uma linguagem inerentemente orientada a objetos; porém, através de um mecanismo de *fallbacks*, é possível implementar vários mecanismos de herança — em particular, o de herança simples, utilizado na implementação do *toolkit*.

Fallbacks são funções chamadas por Lua quando ela não sabe como proceder frente à uma situação de erro. A princípio, cada situação de erro possui um tratamento *default* (uma mensagem de erro é exibida e o programa é terminado); várias situações de erro estão associadas a *fallbacks* específicas, tais como: comparações ou operações aritméticas entre valores não-numéricos, acesso a variáveis não-inicializadas e acesso a índices ausentes em tabelas[3]. Lua permite, através da execução da função `setfallback`, a redefinição de *fallbacks* para rotinas do usuário.

A idéia por trás do mecanismo de herança simples é: dado que um acesso a um campo de uma tabela não obteve sucesso, e dado que esta tabela herda atributos de outra, a tentativa de acesso deve ser repetida na super-classe; o processo se repete enquanto

[2]IUP, por sua vez, se baseou no TEX

[3]"tabela" é como é conhecido um *array* em Lua

houver super-classes para as tabelas em questão, emitindo uma mensagem de erro caso se chegue ao fim da hierarquia sem ter havido sucesso. Deve-se lembrar que este mecanismo de herança se estende também a funções, uma vez que estas também podem ser campos de tabelas. Estas características, utilizadas em conjunto, permitem que utilizemos tabelas como classes, organizando-as de acordo com uma hierarquia e tendo à disposição todo um mecanismo para herança de atributos e métodos.

⟨*Fallback para implementação de herança simples em Lua 61a*⟩ ≡

```
function Index(t, f)
  if f == 'parent' then
    return OldIndex(t, f)
  end
  local p = t.parent
  if type(p) == 'table' then
    return p[f]
  else
    return OldIndex(t, f)
  end
end

OldIndex = setfallback( "index", Index )
```

Trecho raiz

Segundo o trecho de código apresentado acima, sempre que for feito um acesso a um campo inexistente em uma tabela, Lua chamará a função `Index`, passando como parâmetros a tabela e o campo em questão. Caso a tabela possua um campo `parent` (indicando a classe da qual ela é herdeira), o mesmo campo `f` é acessado na tabela indicada por `t.parent`; o processo se repete até que o campo seja efetivamente encontrado ou a tabela em questão não possua um campo `parent`; neste caso, a *fallback* original (armazenada em `OldIndex`) é chamada para prosseguir com o tratamento *default*.

É interessante observar que este mecanismo **não** possibilita controle quanto à acessibilidade de métodos de uma determinada classe por métodos de outra classe (não necessariamente derivada desta), ou mesmo por funções globais da aplicação. Comparando com C++, por exemplo, não há um paralelo com o controle de escopo proporcionado pelos qualificadores *private* e *protected*.

B.4.3 Trecho da implementação do modelo de prioridades

Prioridades se tornam necessárias a partir do momento em que combinamos elementos expansíveis que competem pelo espaço disponível, em uma situação em que uns naturalmente deveriam ter privilégio frente aos outros na distribuição do espaço.

Suponhamos, por exemplo, que um `hbox` possua apenas três filhos: um `fill`, um `canvas` e outro `fill`, nesta ordem. Quando o espaço for distribuído entre estes elementos, é desejável que o `canvas` receba todo o espaço livre disponível, pois ele tipicamente estará

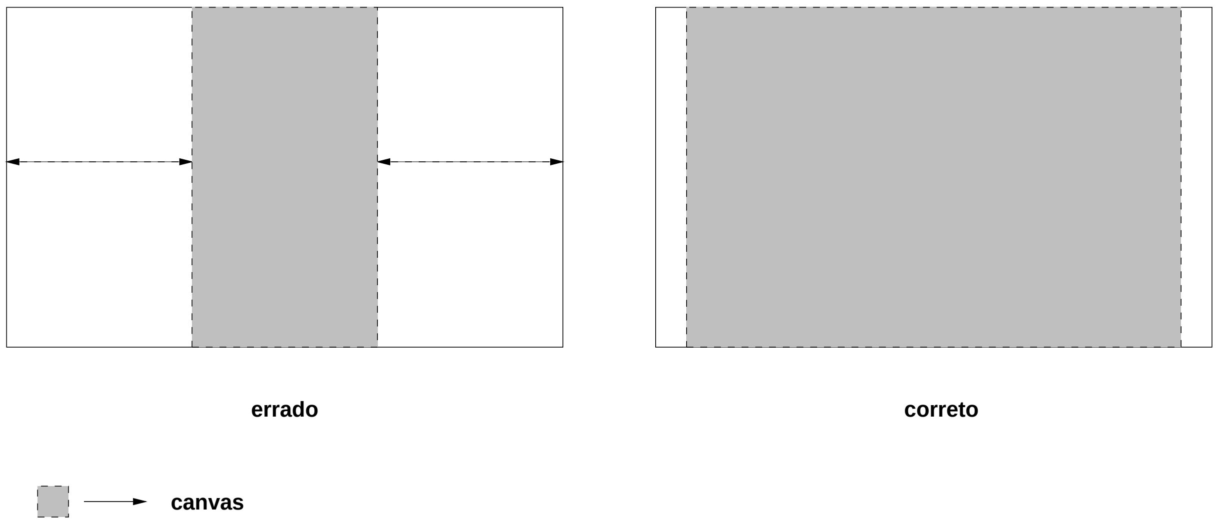

Figura B.1: Distribuição de espaço entre `fills` e `canvas`

sendo utilizado como área de visualização de gráficos, enquanto que o `fill` estará apenas no seu papel de dispor corretamente os elementos no diálogo.
Observação: na figura B.1, notamos que, caso os `fills` não possuam um tamanho mínimo, eles não receberão nenhuma parcela do espaço disponível, possuindo então, ao final da distribuição, tamanho zero — visualmente, será como se eles não estivessem presentes.

A prioridade de um *widget* de composição é igual à maior prioridade dos seus filhos; o espaço a ser distribuído será destinado apenas àqueles que possuírem esta mesma prioridade.

Como valores *default*, a prioridade de um `fill` é zero, e a prioridade de um `canvas` é um — semelhante ao que é feito por IUP; estes valores são atribuídos nos construtores dos elementos em questão. Porém, da maneira como foi implementado o *toolkit*, ele não tem como diferenciar um atributo `priority` definido pelo construtor de uma classe, como um valor *default*, de um outro definido pelo usuário, quando da criação de um objeto. Isto permite que o usuário possa arbitrar prioridades para os objetos que ele instancia — alterando, inclusive, os valores *default*.

Ainda com relação às prioridades, há uma questão que merece atenção: retomemos o exemplo ilustrado na figura B.1; imaginemos agora que o `canvas` em questão possua um máximo na direção x. Se o tamanho disponível para os filhos do `hbox` exceder este limite, a quantidade restante ($total - canvas_{max}$) deve ser distribuída entre os `fills`.

Para que isto aconteça, o algoritmo deve ser capaz de identificar o momento em que o último elemento (ainda incompleto) associado à maior prioridade atinge seu limite, quando deverá "promover" a até então segunda maior prioridade para a condição de primeira prioridade.

O gerenciamento das prioridades foi implementado da seguinte maneira: cada *widget* de composição possui um atributo `priarray`, que é uma tabela indexada por prioridades; cada posição desta tabela possui o número de elementos filhos do *widget* em questão que possuem prioridade igual ao índice. Existe uma prioridade máxima, determinada pela constante `MAX_PRIORITY`:

⟨*Maior prioridade permitida a um elemento 63a*⟩ ≡

```
  MAX_PRIORITY = 5
```

Trecho raiz

Tentativas de se associar uma prioridade maior do que esta a um elemento serão recusadas pelo *toolkit*; por motivos de simplificação, os *widgets* com prioridade zero (*default* para todos os *widgets* com exceção do `canvas`) não são contabilizados. A tabela `priarray` é inicializada em `GROUP:resetsize` (pág. ??), e atualizada em `HBOX:calcmin` (pág. ??) e `VBOX:calcmin` (pág. ??).

Cada *widget* expansível que atinge seu limite em uma das direções informa seu pai do ocorrido; neste momento, o método `delpriority` é chamado (págs. ?? e ??) para atualizar a tabela `priarray`:

⟨*Método GROUP:delpriority 63b*⟩ ≡

```
  function GROUP:delpriority (pri)
    if pri > 0 then
      if self.priarray[pri] > 0 then
        ⟨GROUP → decrementa referência à prioridade 63c⟩
        ⟨GROUP → busca nova prioridade 63d⟩
      end
      ⟨GROUP → informa ao pai da remoção da prioridade 64a⟩
    end
  end
```

Trecho raiz

Este método recebe como parâmetro a prioridade do filho que acaba de atingir seu limite; o que ele faz então é decrementar o contador de referências a esta em `priarray`.

⟨*GROUP → decrementa referência à prioridade 63c*⟩ ≡

```
        self.priarray[pri] = self.priarray[pri]-1
```

Este trecho é usado em 63b

Caso este tenha sido o último filho do grupo associado a esta prioridade, a maior prioridade menor do que ela (e maior ou igual a zero) que esteja associada a algum filho do grupo passa a ser a prioridade do *widget* de composição (e, portanto, referência para a distribuição de espaços).

⟨*GROUP → busca nova prioridade 63d*⟩ ≡

```
        if self.priarray[pri] == 0 then
          local newpri = pri
          repeat
            newpri = newpri-1
          until newpri == 0 or self.priarray[newpri] > 0
          self.priority = newpri
        end
```

Este trecho é usado em 63b

Finalmente, caso o decréscimo do contador de prioridades para a prioridade do filho tenha alterado a prioridade do grupo em questão, devemos informar ao seu pai, para que ele possa incorporar esta mudança.

⟨*GROUP → informa ao pai da remoção da prioridade 64a*⟩ ≡

```
      if self.priority < pri then self[1]:delpriority(pri) end
```

Este trecho é usado em 63b

Para garantir que este processo de remoção de prioridades termine, redefinimos o método `delpriority` da classe `DIALOG` para que ele não faça nada — pois sabemos que o (único) objeto desta classe presente na árvore de elementos será a raiz da mesma, não havendo portanto um pai para o qual propagar o processo.

⟨*Método DIALOG:delpriority 64b*⟩ ≡

```
  function DIALOG:delpriority (pri)
  end
```

Trecho raiz

Bibliografia

[AO90] Adrian Avenarius e Siegfried Oppermann. `FWEB`: A literate programming system for Fortran 8X. *ACM SIGPLAN Notices*, 25(1):52–58, Janeiro 1990.

[AWK] Alfred V. Aho, Peter J. Weinberger, e Brian W. Kernighan. *Awk — A Pattern Scanning and Processing Language*. Bell Laboratories, New Jersey.

[BC90a] Marcus E. Brown e Bart Childs. An interactive environment for literate programming. *Journal of Structured Programming*, 11(1):11–25, 1990.

[BC90b] Marcus E. Brown e David Cordes. Literate programming applied to conventional software design. *Journal of Structured Programming*, 11(2):85–98, 1990.

[Ben86] Jon Bentley. Programming pearls—literate programming. *Communications of the Association for Computing Machinery*, 29(5):364–369, Maio 1986.

[BKM86] Jon Bentley, Donald E. Knuth, e Doug McIlroy. Programming pearls—a literate program. *Communications of the Association for Computing Machinery*, 29(6):471–483, Junho 1986.

[BR93] Michael A. Bell e Mark G. Rivers. Integrating a hypertext interface into a syntax-directed programming environment. Relatório Técnico R93-003, Department of Computer Science, University of Liverpool, Maio 1993.

[Cas95] Carlos Cassino. Trabalho do curso de ambientes de desenvolvimento de software, Julho 1995.

[CB91] David Cordes e Marcus Brown. The literate-programming paradigm. *Computer*, 24(6):52–61, Junho 1991.

[dC96] André Oliveira da Costa. *Um Toolkit para construção de Interfaces com o Usuário*. Grupo de Tecnologia em Computação Gráfica - TeCGraf/PUC-Rio, 1996.

[dOB95] Christiano de Oliveira Braga. Uma ferramenta para geração de documentação de sistemas de software. Dissertação de Mestrado, Departamento de Informática, PUC-Rio, Rio de Janeiro, Brasil, Dezembro 1995.

[Gor95] Tomás Guisasola Gorham. *IUP/Lua Uma interface Lua para o sistema IUP*. Grupo de Tecnologia em Computação Gráfica - TeCGraf/PUC-Rio, 1995.

[Gor96a] T. G. Gorham. Um sistema de depuração para a linguagem de extensão lua. Dissertação de Mestrado, Departamento de Informática, PUC-Rio, Rio de Janeiro, Brasil, 1996.

[Gor96b] Tomás Guisasola Gorham. Um Sistema de Depuração para a Linguagem de Extensão Lua. Dissertação de Mestrado, Departamento de Informática, PUC-Rio, Rio de Janeiro, Brasil, 1996.

[IdFF95] R. Ierusalimschy, L. H. de Figueiredo, e W. Celes Filho. Reference Manual of the Programming Language Lua 2.3. Relatório Técnico 8/95, PUC-Rio, 1995.

[IdFF96] R. Ierusalimschy, L. H. de Figueiredo, e W. Celes Filho. Lua—an extensible extension language. *Software: Practice & Experience*, 26(6):635–652, 1996.

[KL93] Donald E. Knuth e Silvio Levy. *The CWEB System of Structured Documentation, Version 3.0.* Addison-Wesley, Reading, MA, USA, 1993.

[Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, Maio 1984.

[Knu86] Donald E. Knuth. *TEX: The Program*, volume B of *Computers & Typesetting*. Addison-Wesley, Reading, MA, USA, 1986.

[LdFG+96] C. H. Levy, L. H. de Figueiredo, M. Gattass, C. J. Lucena, e D. D. Cowan. IUP/LED: a portable user interface development tool. *Software: Practice & Experience*, 26(7):737–762, 1996.

[LSF94] J. C. S. P. Leite, M. Sant'anna, e F. G. Freitas. Draco-puc: a technology assembly for domain oriented software development. *Proceedings of the 3rd IEEE International Conference of Software Reuse*, 1994.

[McM] Lee E. McMahon. *Sed — a Non-Interactive Text Editor.* Bell Laboratories, New Jersey.

[OC90] Paul W. Oman e Curtis Cook. The book paradigm for improved maintenance. *IEEE Software*, 7(1):39–45, Janeiro 1990.

[OSF91] Open Software Foundation. *OSF/Motif Programmer's Guide*, 1991.

[Ost93] Kasper Osterbye. Literate smalltalk programming using hypertext. Relatório Técnico R93-2025, Institute for Electronic Systems, Agosto 1993.

[Ram92] Norman Ramsey. Literate-programming tools need not be complex. Relatório Técnico CS-TR-351-91, Department of Computer Science, Princeton University, Agosto 1992. (Submitted to *IEEE Software*. Available at `ftp.cs.princeton.edu` in `/reports/1991/351.ps.Z`.).

[RM91] Norman Ramsey e Carla Marceau. Literate programming on a team project. *Software—Practice and Experience*, 21(7):677–683, Julho 1991.

[Sam92] Alan Dain Samples. *User's guide to the M5 macro language*, Setembro 1992.

[Sun90] Sun Microsystems. *Programmer's Overview — Utilities & Libraries*, 1990.

[vAK93] Eric W. van Ammers e Mark R. Kramer. The clip style of literate programming. Relatório Técnico 4.074, Computer Science Department, Wageningen Agricultural University, The Netherlands, Fevereiro 1993.

[Van90] Christopher J. Van Wyk. Literate programming—an assessment. *Communications of the Association for Computing Machinery*, 33(3):361, 365, Março 1990.

[vS93] Arndt von Staa. *Ambiente de Engenharia de Software Talisman, Manual do Usuário.* Staa Informática, 1993.

[Wil92] Ross Williams. Funnelweb user's manual. `ftp.adelaide.edu.au` in `/pub/compression` and `/pub/funnelweb`, University of Adelaide, Adelaide, South Australia, Australia, 1992.

[WS91] Larry Wall e Randall Schwartz. *Programming Perl.* O'Reilly and Associates, 1991.

Uma Ferramenta para Programação Literária Modular

Dissertação de Mestrado apresentada por **Carlos Roberto Serra Pinto Cassino**, no dia 12 de agosto de 1996, ao Departamento de Informática da PUC-Rio e aprovada pela Comissão Julgadora, formada pelos seguintes professores:

Prof. Roberto Ierusalimschy
Orientador
Departamento de Informática - PUC-Rio

Prof. Arndt von Staa
Departamento de Informática - PUC-Rio

Prof. Julio Cesar Sampaio do Prado Leite
Departamento de Informática - PUC-Rio

Visto e permitida a impressão.
Rio de Janeiro, de de 1996.

Coordenador dos Programas de Pós-Graduação e
Pesquisa do Centro Técnico Científico