Semântica Denotacional Legível

José Leite da Silva Júnior
Roberto da Silva Bigonha
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais

Resumo

Este trabalho apresenta uma nova abordagem à definição de linguagens de programação. Partindo do fato que os principais problemas encontrados em definições formais são sua baixa legibilidade e dificuldade de compreensão, foi examinada a adequação da aplicação das técnicas de *literate programming* a semântica denotacional.

O resultado da análise de *literate programming* no contexto de semântica denotacional foi a definição da linguagem de especificação LDS. Essa linguagem une prosa informal e equações semânticas numa mesma estrutura, promovendo um estilo de composição onde a ênfase se encontra na estruturação e apresentação das definições.

[keywords]: linguagens de programação, descrições formais, semântica denotational, LDS, *literate programming*.

1 Introdução

Definições de linguagens de programação têm sido apresentadas em uma das seguintes formas: 1) Basicamente informais, onde a semântica da linguagem é definida através de sentenças em linguagem natural. O formalismo nessas definições não vai além do uso de uma gramática livre do contexto para a apresentação da estrutura sintática da linguagem; 2) Essencialmente formais. A semântica da linguagem é definida através de algum método formal, por exemplo semântica denotacional. Nessas definições há, geralmente, uma ou mais seções dedicadas a uma exposição informal das características semânticas da linguagem. Porém, há pouca ligação entre a definição formal e essa exposição informal.

Os métodos existentes para definição de linguagens de programação apresentam essas características em graus variados. Porém, não parece haver um método que satisfaça plenamente esses requisitos.

A imprecisão semântica, inerente a métodos informais, traz uma série de problemas à descrição através de linguagem natural, que prosa informal é inadequada à descrição de linguagens de programação.

Métodos formais são fortes exatamente onde os métodos informais apresentam problemas, garantindo a precisão e diminuindo as possibilidades de inconsistências em definições de linguagens. Porém, descrições formais também apresentam problemas. A notação especial e os conceitos matemáticos envolvidos, associados à grande quantidade de informação tornam definições formais difíceis de ler, prejudicando a compreensão e a análise das descrições.

Descrições informais têm algumas virtudes importantes. O uso de prosa informal apresenta um forte apelo a intuição, devido a sua naturalidade e facilidade de leitura. Descrições informais também são, geralmente, mais concisas que suas contrapartidas formais. Isso se deve ao fato de definições informais apresentarem descrições parciais. As descrições são completadas através de informação existente no contexto da definição [BGW78].

Chegamos assim a um (aparente) impasse. Definições formais são mais precisas, porém têm menor legibilidade que os métodos informais. Uma possível solução para esse impasse seria a associação de exposição formal à exposição informal, buscando o equilíbrio ideal.

Nós propomos exatamente a associação numa estrutura única do formalismo presente em semântica denotacional à naturalidade da prosa informal [BLAR92]. O equilíbrio dessa forma de exposição resultaria em um estilo de descrição que alcance os objetivos de completeza, clareza, naturalidade, realismo e independência de implementação considerados fundamentais para o sucesso de um método de definição de linguagens de programação. Como previsto em [BJ75] especificações de linguagens de programação consistirão em uma definição formal e uma definição em prosa. As definições poderiam gozar das vantagens dos métodos formais e informais, resultando-se precisas e consistentes, e ainda assim legíveis e naturais.

Assim, definimos a nova linguagem de descrição LDS (*Legible Denotational Semantics*). O projeto de LDS é inspirado em *literate programming* [Knu84]. LDS une em uma mesma descrição dois modos de exposição: um modo informal, permitindo a

apresentação de aspectos da linguagem em linguagem natural, baseado na linguagem de formatação de textos LaTeX [Lam86]; e um modo formal, para formulação das equações denotacionais que descrevem a linguagem, baseado nas linguagens SSL e SDL [Big81].

2 Programação Legível

O termo usado no título desse seção procura capturar a essência do título original: *literate programming*. O fundamental nesse estilo de programação é sua busca de programas mais tratáveis através de maior legibilidade, daí a nossa escolha. Os termos programação legível e *literate programming* serão usados como sinônimos, mas normalmente usaremos o termo original.

Literate programming [Knu84] prega um estilo de programação que priviligia legibilidade. Programas são objetos de publicação, e *literate programming* propõe tratá-los como trabalhos de literatura. Programas têm como público computadores e pessoas, e esse enfoque de literatura conduz a uma visão mais orientada a pessoas no processo de composição de programas. A principal tarefa do programador não é dizer ao computador o que fazer, mas explicar (a outras pessoas) o que o computador deve fazer.

Programação ganha um objetivo mais amplo. Não se deseja "simplesmente" obter um programa correto funcionalmente. O objetivo verdadeiro é obter uma descrição coerente e completa do problema e de sua solução. Os programas não contém apenas código, mas uma especificação do problema, argumentação da solução proposta, avaliação de alternativas e sugestões de modificação [Den87]. Todo o contexto de composição pode ser apresentado para apreciação. Os compromissos do programador são ampliados para abranger clareza de exposição e excelência de estilo. Além da produção de código, o programador deve estar atento à estrutura e organização do código e sua documentação. O principal objetivo é promover uma comunicação mais efetiva entre leitores e escritores, e através de melhor comunicação facilitar a compreensão de programas.

O uso dessa metáfora literária traz ao mundo da programação características comuns em livros. Índices, sumários, tabelas e figuras enriquecem o universo de um programa. A utilização de uma linguagem de formatação de textos possibilita uma apresentação mais atraente. Fórmulas, símbolos matemáticos e diagramas podem ser expressos de forma mais conveniente e uniforme. Índices conferem uma maior liberdade de leitura do programa. O leitor escolhe seus caminhos e recebe do escritor indicações sobre os relacionamentos presentes nas estruturas encontradas durante a leitura.

Um dos efeitos da visão de programas como obras de literatura é um maior destaque da documentação. *Literate programming* aprofunda a idéia que programas devem ser auto-documentados e une numa estrutura única código e texto de documentação. Documentação e código determinam a estrutura de apresentação de programas. O código executável passa a ser um sub-produto do documento de definição do programa. Esse documento contém descrições gerais e discussões intercaladas com definições precisas e fragmentos de código. Programas são escritos nessa mistura de exposição formal e informal, onde métodos formais e informais se completam e se reforçam numa apresentação atraente e elegante [Knu84].

O equilíbrio entre exposição formal e informal permite uma abordagem mais simples à descrição de programas. Em adição, fatos que não podem ser capturados apenas com o uso de código podem ser apresentados informalmente no local que pareça mais apropriado. O próprio processo de desenvolvimento pode ser registrado como documentação útil e facilmente acessível.

Em *literate programming* programas são divididos em seções. Cada seção deve apresentar algum aspecto da solução representada pelo programa. Isso pode incluir prosa e código, ou apenas um deles, o objetivo é expor cada característica do problema da melhor forma possível. A estrutura resultante é mais rica que a simples inserção de comentários em código fonte, pois a documentação tem influência na escolha da estrutura do programa.

Literate programming promove uma maior integração entre documentação e código, enfatizando a importância da documentação. Por um lado, documentação recebe do método a mesma atenção dispensada à codificação, isso promove uma maior consistência entre texto e código, uma vez que manutenção passa a vê-los como um objeto único. Documentação e código ganham uma maior intimidade eliminando ambigüidades e expondo mais claramente seu relacionamento.

A apresentação numa estrutura única e a igualdade entre código e documentação acaba por beneficiar ambos. Código se torna mais claro, ganhando uma estrutura mais adequada à exposição. Documentação se torna mais precisa (há menores riscos de inconsistência), e passa a refletir mais fielmente a realidade da implementação. Ademais, documentar deixa de ser uma atividade extra, e passa a ser executada simultaneamente com codificação. Essa igualdade faz com que a ausência de documentação passe a ser mais evidente que sua presença, estimulando uma documentação mais completa.

Literate programming procura enfatizar duas característica básicas de programas bem escritos: componentes pequenos e fáceis de entender; e pequeno número de conexões lógicas e simples entre esses componentes.

A divisão de programas em seções provém da visão de programas como uma teia de idéias, i.e., seções representam os componentes mais simples de um programa e a estrutura de apresentação determina o relacionamento entre esses componentes. O uso de pequenos segmentos de código é incentivado, com o objetivo de facilitar o entendimento de cada seção da teia de forma relativamente independente das demais. A compreensão do programa pode ser fatorada na compreensão desses pequenos segmentos e seus relacionamentos simples. O uso de pequenas seções de código é benéfico também à construção de programas, pois programadores podem trabalhar em pedaços de código sensíveis, isto é, trechos de código representativos de alguma abstração.

Programas são mecanismos especificando a cadeia de ações que, quando acionadas, produzem algum efeito desejado [Sol86]. Cada techo de um programa em *literate programming* representa um conjunto de ações. Esse conjunto de ações pode ser abstraído e representado através de um nome de seção. As ações podem ser apresentadas, juntamente com a explicação de seu funcionamento, onde for mais apropriado no programa.

A estrutura dos programas é representada como uma malha de mecanismos inter-conectados. Os componentes de um programa e o relacionamento locais são realçados. O acesso à estrutura interna, mesmo de sistemas complexos, é facilitado. A documentação se encaixa nessa estrutura, explicando os mecanismos, os relacionamentos e o próprio processo de composição.

Literate programming procura dar uma maior liberdade na apresentação de programas. Ao invés de restringir a ordem na qual os componentes de um programa devem aparecer, como em muitas linguagens de programação, *literate programming* estimula uma ordem de apresentação adequada à leitura. As seções do programa são apresentadas na ordem considerada mais adequada pelo programador, sem restrições impostas pela linguagem. Essa liberdade de apresentação tem sua correspondente em composição. O autor de programas pode gerenciar a lógica do programa como lhe parecer mais apropriado, sem maiores restrições de organização. Não há necessidade de escolhas definitivas sobre apresentação (e composição) *bottom-up* ou *top-down*, o único compromisso do programador é com lucidez de apresentação.

O uso de seções representa uma alternativa estática à abstração procedimental de linguagens de programação, permitindo que código e documentação sejam relacionados e abstraídos. O uso de seções permite uma forma de apresentação semelhante à técnica de refinamentos sucessivos, possibilitando uma melhor representação dos processos envolvidos na construção do código.

Literate programming torna mais claro o sentido da dupla audiência de programas: pessoas e computadores. Isso lhe confere um certo caráter bilingüe, na mesma linguagem há uma linguagem voltada à formatação do texto, para uma melhor apresentação a pessoas, e uma lingagem para compilação e execução por computadores. Duas ferramentas são utilizadas para dirigir o programa a suas audiências: *tangle* e *weave*. *Tangle* torna o programa adequado ao processamento por computador. O programa resultante tem a ordenação ditada pelo compilador, e está livre de qualquer documentação; pois como seu objetivo é servir aos propósitos de compilação, não há necessidade de explicações informais. *Weave* procura satisfazer as necessidades de legibilidade das pessoas. Seu objetivo é promover maior relevo à estrutura dos programas. Todos os aspectos estilísticos de formatação de documentação e código são considerados para uma melhor apresentação. Em adição, todo o aparato técnico de índices, sumários e referências cruzadas é gerado. O resultado de *weave* é a descrição final do programa da forma mais apropriada à leitura.

2.1 O Sistema `WEB`

O sistema `WEB` de documentação estruturada [Knu83] foi o primeiro sistema a implementar as idéias de *literate programming*. `WEB` provém da união entre a linguagem de formatação de textos TEX e a linguagem de programação Pascal. Um programa `WEB` é composto de uma seqüência de seções (também denominadas módulos), onde cada seção apresenta três partes:

1. uma parte inicial com material explicativo, com objetivo de documentação e escrito em TEX;
2. uma parte de definição de texto para uso em macro-expansões, permitindo abreviar construções que seriam menos intelegívies se escritas por completo cada vez que fossem usadas;
3. uma parte em Pascal, para formulação do código executável do programa.

A ordem dentro de uma seção é sempre essa, porém qualquer das partes pode ser omitida.

A combinação de TEX e Pascal oferece recursos de documentação mais poderosos que o uso em separado das duas linguagens. Por um lado as habilidades tipográficas de TEX permitem uma melhor apresentação do próprio código Pascal e não apenas de sua documentação. Por outro lado código e documentação estão estruturalmente relacionados, evitando problemas de inconsistências e incertezas. A qualidade da documentação também é melhorada, pois como parte integral da implementação do programa, ela pode refletir mais fielmente a realidade da codificação.

As seções de um programa em `WEB` são partes relativamente auto-contidas do programa, projetadas para transmitir uma idéia dentro da lógica geral do programa. Há dois tipos de seção, seções com nome e seções anônimas. Seções anônimas servem como âncoras para o resto do texto do programa. Todo programa tem, pelo menos, uma seção anônima, em torno da qual é determinado o posicionamento das outras seções do programa. Módulos com nome permitem a abstração de algum aspecto do programa e podem ser usadas livremente em outras seções. A documentação presente na definição de uma seção descreve inteiramente o código, porém, o próprio nome da seção deve servir como uma boa descrição do seu conteúdo. `WEB` permite que nomes de seção sejam abreviados, incentivando nomes suficientemente longos para transmitir melhor idéia do conteúdo do módulo, sem com isso impor maior carga (tédio) devido ao uso de nomes muito extensos.

O sistema permite que o programador pense em um nível mais elevado, pois ele deve pensar em termos de definição de programa, não puramente em termos de código. A tarefa "mundana" de tradução da descrição do programa para código executável é deixada a cargo do computador. As seções são apresentadas na ordem que o autor considere mais adequada, sem as restrições impostas pela compilação do código Pascal. Programas são tratados como teias de idéias, um todo que foi

"delicadamente tecido a partir de materiais mais simples" [Knu84]. Essa é a abordagem de WEB à compreensão de programas, deve-se entender as partes mais simples e seus relacionamentos para entender o todo.

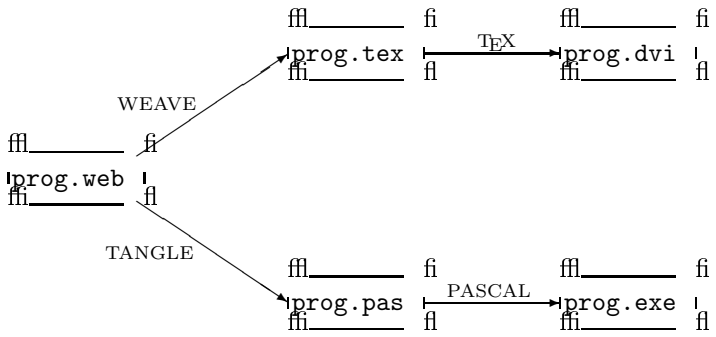

Figura 1: Sistema WEB de Knuth

Tangle e *weave* são originários do sistema WEB. *Tangle* procura produzir a partir da descrição WEB do programa código aceitável por um compilador Pascal. Esse código é construído da seguinte forma: inicialmente, todos os módulos anônimos são agrupados como a aproximação inicial do programa executável, então todas as referências a seções são substituídas pelo código correspondente, por fim as macro-expansões encontradas são realizadas. O código obtido sofre alguma transformações para torná-lo aceitável a um grande número de compiladores, essas são transformações simples, como o truncamento de identificadores muito longos e uso de letras maiúsculas em identificadores e palavras reservadas. *Weave* obtém um texto TEX sintaticamente correto a partir do programa WEB. Convenções de documentação são seguidas para a produção de um documento atraente e legível. Um índice de referências cruzadas, uma tabela de seções e um sumário também são gerados por *weave*. Uma visão geral do sistema WEB é apresentada na Figura 3.1, e detalhes da implementação de *tangle* e *weave* podem ser encontradas em [Knu83].

2.2 Críticas

2.2.1 Problemas

Alguns dos problemas encontrados não são inerentes a *literate programming*, mas específicos do sistema WEB. Esses problemas são discutidos devido a importância de WEB no contexto de *literate programming* e porque WEB serviu como base para o projeto de LDS.

Um dos primeiros problemas encontrados foi o caráter multilingüe de *literate programming*. A notação usada é uma combinação de uma linguagem de formatação de textos e uma linguagem de programação (TEX e Pascal no caso de WEB), alguns comandos próprios da nova notação também são adicionados. Programadores devem se sentir confortáveis e estar preparados para lidar com essa profusão de notações. Erros são inevitáveis no processo de codificação, nesse caso, os erros irão variar de erros na lógica do programa e violações sintáticas ou semânticas a erros de formatação do documento de definição do programa, assim o programador deve dominar as linguagens e entender bem as interações entre elas. A necessidade do aprendizado de uma linguagem de formatação de textos aumenta a carga de conhecimentos necessária ao uso efetivo do sistema, conferindo a *literate programming* um elitismo prejudicial à sua expansão.

No caso específico de WEB, a baixa legibilidade e a dificuldade de escrita de TEX impõem dificuldades de formulação e legibilidade do texto fonte (não formatado) de programas WEB. Isso também traz uma grande distância entre o texto fonte e o texto formatado, com conseqüências de maiores dificuldades na alteração de programas. WEB não oferece mecanismos simples para vencer as regra de formatação de *weave*, assim o programador deve recorrer ao uso direto de TEX para obter o resultado pretendido, essa é uma tarefa, no mínimo, incômoda. A introdução de mecanismos simples de escolha de formato pelo programador pode melhorar bastante esse panorama. Um desses mecanismos simples é a aceitação da escolha de indentação e agrupamento de comandos em linhas feitas pelo programador.

Um problema sutil é originado por uma das virtudes do uso de formatação para produzir a documentação de programas: a atração representada por textos bem formatados pode tentar o programador a buscar maior embelezamento do seu programa, deixando questões funcionais e estrutura sem a devida atenção, essa procura de uma exposição demasiado bela também aumenta a complexidade de programação. A manutenção de programas também pode ser prejudicada, pois alguns programadores relutam em fazer correções em documentos bem formatados, afinal "um texto formatado parece muito bom para estar errado" [Thi86]. Porém, o apelo à estrutura apresentado por *literate programming* minimiza esses problemas, pois a formatação automática de

programas tende a afastar o programador da tarefa de embelezamento de código, e a ênfase em estrutura deixa mais clara a necessidade de correção de código.

Como outros problemas específicos de `WEB` encontramos sua organização de programas, como uma seqüência linear de seções, e a ausência de mecanismos para apresentação, de forma conveniente, de estruturas úteis na explicação de programas, como figuras e tabelas. Embora uma notação hierárquica seja usada na parte de código das seções, o documento como um todo não apresenta essa característica. A forma encontrada em `WEB` de expor alguma hierarquia não é adequada, ou sequer conveniente: seções marcadas com um asterisco são consideradas de maior importância, podem ter um título e são citadas no sumário, as outras seções são colocadas em segundo plano por não terem título ou entrada no sumário. Uma apresentação hierárquica poderia refletir mais fielmente o processo de desenvolvimento e a própria hierarquia existente em programas.

O esquema de macro-expansão também pode trazer dois problemas. O primeiro é relacionado à aparente atomicidade do uso do código de seções. Considere o seguinte exemplo, de [Thi86], onde uma seção é declarada como uma seqüência de comandos e essa seção é usada numa construção que torna apenas o primeiro comando da seção subordinado à semântica dessa construção, como mostrado no seguinte código `WEB`:

```
if (boolean expression) then
    ⟨Section Code_x⟩
```
onde, ⟨Section Code$_x$⟩ ≡ `CMD`$_1$`; CMD`$_2$`;`

Esse problema está relacionado à liberdade sintática presente na composição e uso de seções, Soluções para esse problema devem envolver alguma forma de análise sintática, porém, não devem ser impostas restrições à composição. O segundo problema do uso de macro-expansão é a possibilidade de difusão de erros pelo programa de uma forma sutil e efetiva. Dessa vez, a ausência de localidade na definição de seções é a origem do problema. A incorporação de localidade no código de seções pode ser examinado como solução.

O enfoque (algumas vezes) excessivo no caráter literário de programas pode causar alguns inconvenientes. Textos literários apresentam material bem conhecido e estável, assim, sua apresentação pode receber maior atenção. Programas, geralmente, sofrem contínua modificação e extensão. Novas versões rapidamente tornam as antigas obsoletas; dessa forma, a apresentação não ser tão cuidada em momentos turbulentos do desenvolvimento, onde a possibilidade de alterações é grande. Se a metáfora literária é usada criteriosamente nenhum problema surgirá, de outra forma, o trabalho de atualização de versões pode consumir muito esforço.

Finalmente, sistemas de *literate programming* que usam linguagens de formatação de textos como TEX não aderem muito bem ao processamento interativo, dificultando seu uso em sistemas interativos de desenvolvimento de *software*. O custo adicional de processamento e produção da documentação poderia, a princípio, ser considerado uma grande desvantagem, mas acreditamos que a qualidade da documentação produzida contrabalança esses custos através da maior facilidade de compreensão de programas.

2.2.2 Vantagens

A filosofia (e o próprio título) de *literate programming* indica um maior realce na necessidade de legibilidade de programas, destacando a importância de considerar programas como forma de comunicar a intenção de programadores. A audiência humana de um programa passa a receber maior atenção. Programas deixam de ser simples conjuntos de instruções que orientam um computador na execução de tarefas, para serem descrições gerais de um problema e sua solução. Programas são feitos com a intenção de serem lidos e executados, não somente executados.

A apresentação do programa reflete melhor o encadeamento de idéias que levou ao seu desenvolvimento. Como código fonte é muitas vezes a única documentação confiável de um programa, *literate programming* procura tornar código fonte mais legível e assim tornar programas mais fáceis de compreender e tratar.

O Programador não escreve apenas um programa comentado, mas um artigo sobre o programa e seu processo de desenvolvimento e *literate programming* garante que o artigo pode ser executado. Essa característica é denominada verossimilhança em [vW90], e considerada fundamental para *literate programming*. Documentação é parte integral do programa, incentivando programas mais documentados. Há mesmo uma melhora qualitativa na documentação, devido ao maior destaque recebido e a maior intimidade com o código.

Literate programming apresenta vantagens sobre o uso de comentários em linguagens de programação, uma vez que, o estilo de documentação apresentado nas seções de um programa deixa claro um relacionamento sintático e semântico entre o código e sua documentação. No caso de código e comentários, esse relacionamento é puramente semântico, assim o programador deve entender o programa suficientemente bem para entender os comentários e assim torná-los úteis.

A integração código/documentação e a ênfase em características estilísticas (como o uso de diagramas) tornam a documentação não só mais acessível, mas também mais atraente e útil. O método em si não pode garantir a consistência entre código e documentação, afinal consistência é um compromisso também do programador. Porém, o código é considerado quase um sub-produto da documentação, e esse vínculo reduz bastante as chances de inconsistências. Há mesmo uma ênfase para que

a estrutura do programa seja o resultado de considerações de estrutura de código e documentação. A documentação das seções representa um nível mais alto de abstração e pode transmitir informações que não são facilmente obtidas através do código.

Programas são produzidos a partir de pequenos segmentos de código e documentação. Cada segmento representa um mecanismo e pode ser abstraído através do nome da sua seção. Os mecanismos de um programa são ordenados segundo a vontade do seu autor. O programador tem a liberdade de codificar e apresentar o programa na ordem que considere mais adequada ao seu entendimento. Livre de restrições de ordenação impostas pela linguagem, o programador pode buscar melhor estilo e riqueza de estrutura. O uso de pequenos segmentos de código parece estar de acordo com a forma natural de composição de programas [Wei84]. Assim, o leitor pode esperar que o programa reflita as idéias do autor. O conjunto de índices propicia liberdade de navegação sobre o código, o leitor escolhe seus caminhos e recebe, através dos índices, a orientação do autor sobre os possíveis novos caminhos a seguir.

O uso do mecanismo de seções pode ajudar a focalizar melhor as atenções do programador. Consedere novamente o exemplo de [Thi86]: Um programador escreve uma rotina para executar uma tarefa `T`, no caso de `T` ser simples, a adição de checagem e manipulação de erros faz com que a rotina pareça uma rotina de recuperação de erros. Um bom programador omitirá ou abreviará a checagem de erros fazendo a rotina parecer como deve. *Literate programming* torna a manipulação de erros um único comando (o nome de uma seção), assim o programador pode concentrar sua atenção em cada parte do código separadamente, dessa forma, cada parte da rotina terá maior possibilidade de codificação completa, sem prejuízos a sua aparência.

A visão de programas como obras de literatura traz outras vantagens. Os leitores poderão experimentar um gosto prático (dificuldades de implementação) de idéias algorítmicas, tornando mais adequado o processo de entendimento e avaliação dessas idéias. Em adição, programadores se colocam numa situação de exposição, esforçando-se em apresentar suas idéias da melhor forma possível. Como conseqüência, essas idéias ficam mais claras para ele mesmo, ajudando a encontrar erros e avaliar sua extensão mais facilmente.

Finalmente, devemos salientar a flexibilidade de *literate programming*, tanto pela sua aplicabilidade a um amplo conjunto de linguagens, por exemplo, Pascal [Knu84], C [Thi86], FORTRAN [AO90], Ada [WB89] ou Smalltalk [RS89], como pela sua liberdade de composição e leitura de programas. Como observado em [vWHG87], "caráter literário em programação tem diferentes significados em diferentes circunstâncias. Não é uma questão simplesmente de arte ou eficiência; é uma questão de adequação de contexto". Acreditamos que *literate programming* aborda a composição de programas de forma adequada, sendo um passo inicial na direção da representação de programas numa estrutura mais rica.

Literate programming tem como base a idéia que programas devem ser orientados à leitura por pessoas. Reunindo muitos princípios (equilíbrio de notação formal e informal, auto-documentação de programas, ênfase em estrutura, estilo de programação e formatação de programas) numa estrutura simples e uniforme, *literate programming* busca fornecer os meios para produção de programas de melhor qualidade.

Devemos salientar a necessidade de sistemas como `WEB`. Os serviços oferecidos aliviam a carga de programação e incentivam o uso de recursos que facilitam a leitura de programas. Os problemas apresentados na Seção 3.5 impossibilitam qualquer tentativa de uso de *literate programming* sem ferramentas automáticas.

Semântica de linguagens de programação tem como objetivo básico a exposição, assim, acreditamos que vantagens poderiam advir do emprego de *literate programming* a semântica denotacional. Tendo o sistema `WEB` como ponto inicial, definimos a linguagem LDS (uma integração de texto e equações). Como em `WEB`, LDS é dirigida a suas audiências através dos programas *tangle* e *weave*.

3 A Linguagem LDS

LDS (*Legible Denotational Semantics*) se propõe a resolver os problemas de legibilidade de semântica denotacional através do uso de modularidade e a agregação de um modo informal de exposição. LDS é o resultado da união de uma linguagem de formatação de documentos e duas linguagens de definição formais (uma linguagem de especificação sintática e uma linguagem modular de especificação semântica). O projeto de LDS está baseado em considerações sobre o sistema de documentação `WEB` e os detalhes de seu projeto estão em [Lei93]. Procuramos eliminar, ou pelo menos amenizar, os problemas encontrados em `WEB` e tirar o máximo proveito de suas virtudes.

Uma diferença evidente entre LDS e `WEB` é a presença de uma estrutura hierárquica na documentação. Enquanto `WEB` apresenta apenas um nível de seções, LDS apresenta uma estrutura aninhada que permite a definição de até cinco níveis de seções. Os aspectos da definição podem ser apresentados numa hierarquia de capítulos, seções, subseções, etc. mantendo a referência a textos literários, enquanto sugerem a estrutura presente na definição. Equações podem ser apresentadas em todos esses níveis e usadas em qualquer ponto da definição. O sumário apresentado junto a documentação provê uma imagem geral da estrutura e organização da definição.

A própria estrutura interna das seções sofreu alterações. LDS não tem a capacidade de definição de texto para macro-expansão. Acreditamos que essa característica seria de pouca utilidade na descrição de linguagens, porém, efeito semelhante pode ser obtido através da definição e uso de seções de código (equações). Uma seção é composta de uma parte informal seguida

de uma parte formal em SSL ou SDL. Qualquer dessas partes pode ser omitida a critério do autor da definição. Essa estrutura estabelece um vínculo sintático entre a parte formal e a informal, tornando seu relacionamento mais evidente.

O uso de uma estrutura mais flexível também foi considerado. Nessa estrutura, as seções poderiam ser compostas de texto informal e formal em qualquer ordem e sem limites para o término de uma seção (em LDS, há no máximo uma parte informal e uma parte formal por seção). Descartamos esse formato pois a relação entre os trechos formais e informais se tornaria puramente semântica, quando nós desejávamos uma estrutura que deixasse mais claro esse relacionamento. Como em `WEB`, a parte formal pode receber um nome, permitindo a abstração de seu conteúdo em outros contextos da definição. Não restringimos o uso dessas seções de código, pois um dos objetivos do emprego de *literate programming* é conceder maior liberdade de composição, mas devemos registrar que seções devem ser representativas de alguma abstração e portanto devem ser usadas com critério.

A liberdade sintática presente em `WEB` foi mantida. [CB91] sugere uma restrição interessante à codificação de seções: toda função, procedimento ou módulo iniciado numa seção deve terminar naquela mesma seção. Com isso as dependências entre as seções da definição seriam mais simples e cada seção poderia ser vista como uma unidade funcional completa. Porém, acreditamos que essa restrição iria de encontro a uma das bases de *literate programming*: a liberadade de composição. Assim preferimos manter a formulação de seções livre de restrições e deixar como sugestão aos autores de definições a busca por interações simples entre os módulos e entre as seções da definição.

A formatação do documento de descrição também apresenta diferenças. Foi incorporado um mecanismo para formatação de trechos da documentação como se fossem código. Isso permite o uso de equações na documentação com propósitos ilustrativos, mantendo a consistência com a formatação das equações que realmente integram a parte formal, sem impor cargas ao autor da definição. Essa característica é sugerida em [WB89]. O escritor também tem maior controle sobre a formatação do código. Suas escolhas de indentação e agrupamento de expressões são mantidas no documento formatado. Porém, restringimos os comandos da linguagem de formatação que podem ser usados na parte formal de cada seção. Nosso objetivo foi manter um maior controle do sistema sobre o processo de formatação. Há comandos específicos de LDS que permitem ao escritor influenciar no resultado da formatação.

Uma definição LDS pode ser dividida em diversos arquivos. Essa característica tem implicações tanto na produção da descrição quanto na produção do compilador da linguagem. O escritor não é obrigado a escrever a descrição como apenas um grande arquivo, ele pode dividir a descrição em um conjunto de pequenos arquivos, e agilizar a tarefa de edição. Por outro lado, isso abre possibilidades para a compilação em separado das equações denotacionais que produzirão do compilador da linguagem. O resultado é uma melhor abordagem à composição de definições extensas.

A possibilidade de divisão também dá origem a uma restrição: o código de um módulo SDL deve ser totalmente definido em apenas um arquivo, pois a parte formal das seções LDS é visível somente no arquivo onde a seção foi definida. Essa restrição também é válida para SSL; assim, como a gramática SSL é composta de apenas um módulo, a definição sintática da linguagem deve estar totalmente concentrada em um arquivo. A possibilidade de tornar definições de seções de código visíveis externamente a seus arquivos traria dificuldades de composição de definições, e mesmo na implementação de LDS, pois a liberdade sintática apresentada nessas seções torna seus limites e interfaces menos precisos que, por exemplo, módulos SDL.

A divisão da definição em arquivos torna o processo de obtenção da documentação e do compilador mais complexo, como ilustrado na Figura 5.1. A aplicação de *weave* a algum dos arquivos da definição produz um trecho da definição, o processo de *thread* une os arquivos para posterior formatação, pelo processador LaTeX, que produz o texto final da definição. *Tangle* funciona de forma similar, extraindo os módulos SSL e SDL apresentados em cada arquivo. Os compiladores SSL e SDL transformam esses módulos no tradutor e no gerador de código (implementação da semântica), cujo conjunto é o compilador da linguagem. Os processos *weave*, *thread* e *tangle* são discutidos no Capítulo 5.

Nas próximas seções são apresentados informalmente LaTeX, SSL e SDL (as linguagens que formam a base de LDS). Os comandos específicos de LDS são discutidos ao final.

3.1 Formatação de Definições

O principal propósito da composição de documentos é a apresentação de idéias. O documento é a representação concreta das idéias de seu autor [PJN85], e a estrutura do documento é determinada pelo pensamento do escritor. Essa estrutura lógica deve ser demonstrada visualmente, não podemos considerar documentos como simples seqüências de palavras agrupadas em linhas e páginas. A existência de componentes lógicos (sentenças, seções, figuras, etc.) deve ser compreendida e as linguagens de formatação de texto devem reconhecer e manipular esses componentes de forma a apresentar os documentos de acordo com sua estrutura.

A linguagem de preparação de documentos LaTeX é uma versão especial do programa de formatação TeX [Knu86]. TeX foi projetado para produção de textos de alta qualidade gráfica, possibilitando ao leitor o posicionamento dos elementos de um texto com precisão. Essa concentração no arranjamento de objetos no documento dá origem a uma linguagem flexível e poderosa, porém, difícil de usar. A utilidade de um sistema de formatação está fortemente ligada à sua linguagem de especificação de documentos, dessa forma, LaTeX com sua ênfase em concisão e simplicidade, torna mais acessível toda a capacidade tipográfica de TeX.

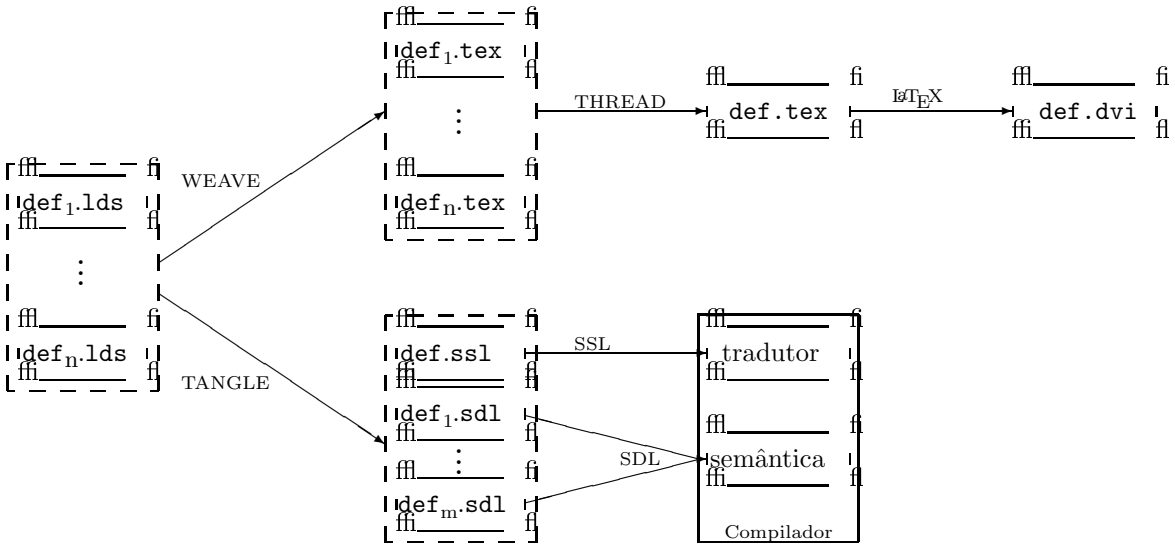

Figura 2: WEB semântico

Documentos apresentam uma estrutura lógica, e LaTeX provê os meios para distinguir visualmente os diferentes elementos dessa estrutura. LaTeX é um formatador (ou compilador) de documentos, isso significa que ele aceita como entrada uma descrição do documento, preparada em algum editor de textos. A descrição do documento contém o texto a ser apresentado e indicadores da estrutura desse texto. Através dessa indicação de estrutura, LaTeX pode fazer uso de elementos visuais durante o processo de formatação, com o objetivo de obter um documento onde a estrutura seja mais explícita.

[Lam86] caracteriza a abordagem de formatação de LaTeX como projeto lógico. De fato, os comandos e ambientes da linguagem procuram principalmente descrever a lógica de composição de documentos. Essa abordagem está em oposição aos formatadores interativos (ou interpretadores) de documentos. Formatadores interativos permitem a visualização do resultado da formatação durante o processo de composição, o usuário desses sistemas manipula diretamente a representação visual do documento. Esse processamento interativo é bastante conveniente, porém, o apelo que a forma do documento representa pode por em perigo o processo de composição estrutural. O projeto lógico, por outro lado, encoraja a ênfase em estrutura e não em resultados visuais imediatos, permitindo que o autor se concentre nos aspectos relevantes, i.e., o conteúdo do documento e não sua aparência. Todavia, LaTeX reconhece a importância que excelência tipográfica pode representar e coloca à disposição do usuário comandos capazes de manipular com precisão o processo de formatação de documentos.

Literate programming enfatiza estrutura de programas, assim, é natural buscarmos uma linguagem de formatação que incentive estruturação de documentação. TeX e LaTeX satisfazem a necessidade de estruturação. A preferência por LaTeX se deveu a sua maior facilidade de uso (sem perda de capacidade de formatação). LaTeX deixa à disposição do autor de definições em LDS um conjunto de poderosos ambientes e comandos de formatação, numa notação sóbria e legível.

Documentos podem ser arbitrariamente longos, assim LaTeX permite a divisão do texto em vários arquivos. Há também a possibilidade de definição de novos comandos e ambientes a partir de comandos e ambientes já existentes. Em resumo, LaTeX atende às necessidades de formatação de documentos através de sua capacidade tipográfica. Em adição, oferece uma linguagem de definição de documentos razoavelmente legível e fácil de usar. [Lam86] apresenta uma ampla descrição de LaTeX.

3.2 A Linguagem de Especificação Sintática

Uma notação para descrição das características sintáticas é necessária por dois motivos: em primeiro lugar, um compilador será gerado automaticamente, então é necessária uma descrição sintática da linguagem para produção do tradutor. Em segundo lugar, a descrição sintática é indispensável a uma descrição completa da linguagem.

A descrição sintática de linguagens é considerada bastante mais fácil de apresentar do que sua descrição semântica, e métodos baseados em gramáticas livres do contexto têm sido utilizados com sucesso. Segundo [Pag81] há quatro aspectos importantes relacionados à sintaxe de linguagens de programação: 1) sintaxe abstrata; 2) sintaxe concreta; 3) correspondência entre sintaxe abstrata e concreta; 4) aspectos sensíveis ao contexto.

A sintaxe abstrata pode ser vista como a definição fundamental de uma linguagem. A sintaxe abstrata especifica a estrutura de composição de programas, determinando o tipo e os componentes de cada estrutura da linguagem. Apenas os aspectos semanticamente relevantes são tratados. De fato, a sintaxe abstrata determina as construções que serão envolvidas na definição semântica da linguagem, i.e., as estruturas que terão associadas funções semânticas determinando seu significado. A sintaxe abstrata desempenha um papel fundamental em definições denotacionais, como observado em [Big81]: "a sintaxe abstrata pode

ser vista como um sistema de equações de domínios, cuja solução especifica o domínio sintático das construções da linguagem".

A sintaxe concreta da linguagem determina a estrutura de representação das construções da linguagem. Nela está codificada toda a informação necessária para a análise sintática determinística de programas. A sintaxe concreta representa uma realização textual da forma fundamental de um programa, i.e., a sintaxe concreta pode ser vista como uma implementação da sintaxe abstrata da linguagem. A correspondência entre a sintaxe abstrata e a concreta deve ser formalizada em definições completas de linguagens, com o objetivo de deixar claro o papel estrutural da sintaxe concreta.

As condições sensíveis ao contexto são comumente denominadas de semântica estática e representam restrições à forma que determinados trechos do programa podem apresentar. Essas restrições são originadas em mútuas dependências entre as partes do programa, e.g., a declaração e uso de identificadores. Embora gramáticas livres do contexto (GLC) sejam comumente usadas em descrições sintáticas, elas não são suficientes para descrever aspectos sensíveis ao contexto. Essas características podem ser descritas através de extensões a GLC (e.g., YACC) ou através de métodos de descrição semântica. [Big84] apresenta um método para definição da semântica estática através do uso de semântica denotacional direta.

A linguagem de especificação sintática em LDS é SSL (*Syntax Specification Language*). SSL é basicamente uma linguagem para formulação de gramáticas livres do contexto. SSL apresenta um notação semelhante a BNF, porém, SSL permite especificar separadamente os aspectos léxicos da linguagem, i.e., o reconhecimento das suas unidades textuais (*tokens*), e a correspondência entre a sintaxe abstrata e a sintaxe concreta de uma linguagem.

Apenas linguagens livres do contexto podem ser descritas em SSL. Isso significa que não é possível especificar em SSL os aspectos sensíveis ao contexto de uma linguagem de programação. A abordagem sugerida ao usuário de SSL para descrição desses aspectos é a apresentada em [Big84]. SSL se encaixa num sistema para descrição completa (sintaxe e semântica) de linguagens de programação, assim, não há qualquer problema nessa limitação de SSL.

Uma descrição SSL está dividida em três partes: `SYNTAX`, `DOMAINS` e `LEXIS`. Estas partes determinam, respectivamente, os aspectos sintáticos, os domínios sintáticos e os aspectos léxicos da linguagem. Os domínios sintáticos são declarados através da associação de listas de símbolos não terminais a um identificador que representa a categoria sintática da qual os símbolos fazem parte. As especificações sintática e léxica têm basicamente a mesma forma, i.e., uma gramática livre do contexto. Essas produções indicam as regras de substituição de símbolos para composição de programas na linguagem.

Há duas vantagens na separação da apresentação dos aspectos léxicos e sintáticos. Primeiramente, apenas uma notação é usada, sem impor problemas de visualização da estrutura sintática da linguagem. A segunda (e principal) vantagem se manifesta na geração do tradutor da linguagem. O algoritmo de análise é guiado por uma tabela cujo tamanho cresce rapidamente com o número de símbolos da gramática. Assim, essa divisão tem como efeito a redução no tamanho das tabelas usadas. Devemos notar ainda que o tempo de execução dos analisadores produzidos também é diminuido através dessa divisão.

SSL também procura simplificar a especificação de `LEXIS`. SSL adiciona automaticamente produções para o reconhecimento dos símbolos terminais especificados em `SYNTAX` cuja formação não seja indicada nas cláusulas de `LEXIS`. Essa característica leva à adição de um (pseudo-)operador às expressões usadas nas produções da gramática. O operador `OUT` permite a indicação que um símbolo é produzido pelo analisador léxico para comunicação com o analisador sintático. Esse operador indica a SSL que não deve ser gerada uma alternativa para o reconhecimento desse símbolo.

SSL apresenta duas construções bastante úteis na formulação de descrições sintáticas. Essas construções são um tipo especial de produção e operadores de iteração. A produção especial é uma forma mais simples de produção, que permite a especificação conveniente de produções compostas de símbolos terminais. Os operadores de iteração correspondem a uma forma restrita da operação de fecho de Kleene, e determinam a repetição do símbolo ao qual estão ligados. Esses operadores representam uma alternativa ao uso de recursão, permitindo um menor número de símbolos não terminais e uma notação mais compacta.

Como dissemos, a especificação sintática de linguagens deve envolver a descrição da sintaxe abstrata e a sua correspondência com a sintaxe concreta. SSL deve oferecer mecanismos para descrição da sintaxe abstrata não apenas devido a questões de completeza da definição, mas também porque a sintaxe abstrata é essencial à formulação da descrição semântica da linguagem, pois a sintaxe concreta é cheia de detalhamentos que tornariam obscura uma descrição semântica baseada nela.

SSL dispõe de meios para especificação da sintaxe abstrata e foi projetada para tornar clara o relacionamento entre as duas formas de sintaxe. A sintaxe abstrata e sua relação com a sintaxe contreta é estabelecida através da associação de expressões com as produções da gramática. As regras da gramática indicam a estrutura concreta da linguagem, enquanto as expressões determinam como deve ser construída a árvore de derivação baseada em sintaxe abstrata. Em SSL, a sintaxe abstrata determina a construção das árvores de derivação, e essa representação estrutural intermediária de programas é usada na definição semântica da linguagem.

Em resumo, especificações SSL determinam um mapeamento entre a representação concreta de programas (uma seqüência de caracteres) e uma representação abstrata (nodos de árvore de derivação). A sintaxe concreta é descrita através de regras gramaticais numa forma semelhante a BNF. Expressões podem ser associadas a essas regras, estabelecendo a sintaxe abstrata da linguagem (e seu relacionamento com a sintaxe concreta). Esse mapeamento é descrito em dois passos: primeiramente programas tratados como seqüências de caracteres são transformados em uma lista dos elementos básicos da linguagem (através da especificação de `LEXIS`), então essa lista é mapeada segundo as construções da linguagem em nodos da árvore de derivação (através da especificação de `SYNTAX`). O projeto de SSL está basedo na linguagem GRAM (descrita em [Mos78]), e [Big81]

apresenta a especificação completa de SSL, bem como sua definição formal.

3.3 A Linguagem de Especificação Semântica

É desejável que uma linguagem para especificação de semântica denotacional mantenha ao máximo a notação desenvolvida por Scott e Strachey [SS71] comumente usada em semântica denotacional. Essa notação é poderosa o suficiente para descrições precisas de linguagens de programação, e a proximidade com ela permite uma transição mais fácil entre definições nessa notação e definições na linguagem. Porém, há alguns aspectos da notação de Scott e Strachey difíceis de implementar, assim, conveniências de notação devem ser adicionadas para permitir o processamento por computador.

A linguagem de especificação semântica de LDS é SDL (*Semantic Definition Language*). SDL mantém a notação de Semântica denotacional e seu desenvolvimento abordou aspectos como modularização, controle de visibilidade e polimorfismo de tipos, através de uma total formalização da notação empregada.

A estrutura modular de SDL obedeceu aos seguintes princípios de projeto (originalmente orientados ao projeto de linguagens de programação):

1. A linguagem deve oferecer construções que permitam a composição de definições como a definição e agregação de unidades menores (módulos);

2. O mecanismo de modularização deve permitir controle amplo e conveniente sobre a visibilidade dos objetos definidos em um módulo;

3. Deve haver uma estrutura hierárquica de módulos. Assim, definições poderiam apresentar uma estrutura mais visível, através da definição de módulos em vários níveis. O simples aninhamento de módulos é o candidato natural ao estabelecimento dessa hierarquia. Em LDS, os módulos SDL são restritos a um nível apenas.

4. A linguagem deve permitir a definição de módulos em duas partes: a parte de definições, que estabelece a interface do módulo com o mundo externo, e a parte do corpo, onde as entidades do módulo são efetivamente definidas.

Em SDL há três tipos de módulo: um módulo principal, módulos secundários externos e módulos internos. O módulo principal contém uma expressão final que define o significado de toda a descrição semântica. Módulos externos são divididos em duas partes: a parte da definição, onde os elementos exportados (visíveis no exterior do módulo) são declarados e os elementos importados de outros módulos são identificados; e a parte do corpo, onde os domínios e funções do módulo são implementados. Módulos internos têm uma estrutura semelhante à dos módulos externos, porém, estão sujeitos a regras especiais de escopo.

A função básica de um módulo externo é permitir o agrupamento de funções, domínios e outros valores relacionados, para posterior uso em outros módulos. O controle de visibilidade, através da especificação de listas de exportação e importação, permite uma seleção criteriosa dos serviços que um módulo oferece a outros, bem como dos serviços utilizados a partir de sua definição em outros módulos. Detalhes de implementação podem ser confinados ao corpo do módulo, enquanto o serviço pode ser utilizado externamente através da descrição de sua interface na parte de definição do módulo.

Módulos internos são permitidos apenas no corpo do módulo. Esses módulos internos são visíveis apenas localmente, através de cláusulas de importação do corpo de um módulo, que somente podem especificar importação de entidades definidas em módulos internos. O aninhamento de módulos é a alternativa oferecida por SDL à composição de definições como uma coleção "amorfa" módulos externos [Big81]. Porém, em LDS há somente um nível de módulos, i.e., os módulos internos foram eliminados. O uso de apenas um nível de módulos parece estar de acordo com as idéias de *literate programming*, onde há ênfase em relacionamentos estruturais e não na apresentação de uma possível hierarquia de módulos. Ademais, módulos internos não são comumente usados, são redundantes em relação ao uso de cláusulas de importação/exportação, e poderiam produzir estruturas mais complexas.

O formato dos módulos SDL tem como objetivo deixar claras as intenções do autor quanto à estruturação da definição. A notação torna explícitas as mútuas dependências e interfaces de módulos. A estrutura e regras de escopo dos módulos internos são apresentados em [Big81]. Módulos internos não são presentes em LDS, então não mais consideraremos essa característica de SDL.

[Big81] enfatiza os benefícios de compilação em separado. Compilação independente é descartada, pois a checagem de consistência de tipos é considerada fundamental. Compilação em separado permite uma grande flexibilidade na compilação de módulos, sem prejuízos à checagem de tipos, pois a interface entre os módulos é sempre exigida. A parte de definição dos módulos SDL fornece a informação necessária à checagem de consistência de tipos. A implementação (corpo) dos módulos somente é necessária durante a fase de ligação, onde o código final do compilador é gerado.

Outra característica interessante de LDS é a possibilidade de definição de funções capazes de manipular entidades de uma ampla variedade de tipos (domínios). O usuário de SDL pode definir funções cujos parâmetros têm tipos diferentes em diferentes aplicações. A disciplina de tipos de SDL determina que todos as questões relacionadas a tipos de entidades sejam resolvidos durante a compilação da definição, assim informação de tipos será dispensável durante a sua "execução". SDL procura aliar as vantagens de uma linguagem fortemente tipada à flexibilidade da definição e uso de funções polimórficas.

Devemos concluir enfatizando a proximidade entre a notação usada em SDL e a notação "padrão" de semântica denotacional. Os benefícios dessa proximidade se encontram principalmente na maior facilidade de tradução entre definições nessas notações e na facilidade de aprendizado de SDL por conhecedores de semântica denotacional. Como extensão à notação padrão, SDL oferece um mecanismo de modularização que torna as definições semânticas mais legíveis e fácies de formular e depurar. A descrição completa de SDL e sua definição formal se encontram em [Big81]. [Rod93] apresenta os detalhes da geração de compiladores a partir de uma definição semântica em SDL.

3.4 Os Comandos LDS

Além dos comandos LaTeX, SSL e SDL, a linguagem LDS agrega alguns comandos próprios. A adição desses comandos tem o objetivo de deixar clara a estrutura da definição (refletindo a estrutura na documentação), preencher algumas necessidades de formatação oferecidas de forma pouco conveniente em LaTeX, e produzir um formato mais homogêneo na documentação final. Procuramos, todavia, manter o conjunto de comandos o mais reduzido possível, pois a adição de um grande número de comandos poderia trazer complicações à interação das linguagens que compõem LDS.

Os comandos indicam estrutura, modo de exposição, apresentação das definições e orientam a produção de texto secundário. Texto secundário é toda informação adicional produzida (semi-)automaticamente, cujo objetivo é oferecer uma melhor perspectiva do e acesso mais fácil ao texto da definição.

Weave produz quatro tipos de texto secundário: 1) um sumário; 2) um índice geral; 3) um índice de módulos; 4) um índice de seções. *Weave*, por convenção, insere essas informações em toda definição LDS. As definições produzidas apresentam essas estruturas na seguinte ordem: sumário, texto da definição (arquivos LDS), índice geral, índice de módulos e índice de seções. Os comandos LDS permitem a alteração do texto secundário segundo critérios do autor da definição.

3.5 Definições em LDS

Na seção anterior apresentamos uma visão geral da estrutura de LDS. O caráter multi-lingüe de LDS ficou estabelecido através da descrição das linguagens que a compõem: LaTeX, SSL, SDL e comandos próprios. As linguagens foram descritas de acordo com suas definições originais, porém, algumas alterações e restrições foram introduzidas na forma como essas linguagens são usadas em LDS. O nosso objetivo foi manter a união mais simples e uniforme. A maior parte das alterações realizadas não tem grande influência na estrutura original das linguagens, assim os comentários a respeito dessas linguagens ainda são válidos.

Inicialmente observamos as alterações no uso da linguagem LaTeX. As alterações são restrições de LDS ao uso de comandos LaTeX. Em primeiro lugar, alguns comandos LaTeX não são permitidos em LDS. Comandos de inserção de arquivos (`\include`, `\includeonly`, `\input`) não são permitidos em definições. O objetivo dessa restrição é assegurar que o texto produzido por *weave*/LaTeX é o mesmo texto usado na produção do compilador da linguagem (por *tangle* e os compiladores SSL e SDL). A função desses comandos é a divisão de um texto em diversos arquivos, e isso é possível em LDS, assim não há qualquer perda com essa restrição. Os comandos de entrada e saída em terminal `\typein` e `\typeout` não devem ser utilizados em LDS, pois o ambiente é responsável pela interação com LaTeX, e isso pode trazer dificuldades na interface.

A última (e maior) restrição é a proibição do uso de comandos de LaTeX na parte formal das seções. Apenas os comandos de início (ou proibição) de nova linha (`\\`, `\newline`, `\linebreak` e `\nolinebreak`) e nova página (`\clearpage`, `\cleardoublepage`, `\pagebreak`, `\nopagebreak`, `\samepage` e `\newpage`) podem ser usados nessa parte formal. *Weave* cuida da formatação das equações e há um conjunto de comandos LDS para controlar essa formatação, dessa forma, não há maior necessidade de dispor de todo o poder de formatação de LaTeX. A restrição tem como objetivo simplificar a tarefa de formatação, de forma que *weave* possa manter estrito controle sobre esse processo sem entrar em detalhes do funcionamento de LaTeX.

A linguagem SSL sofreu duas alterações: 1) o símbolo inicial da gramática deve ser explicitamente indicado através do operador `&start`; 2) a notação de tuplas "`<...>`" foi eliminada. Em SSL, há duas formas de especificação de tuplas: "`<...>`" e "`(...)`", a segunda notação foi mantida. A primeira alteração tem como objetivo deixar mais claro ao leitor da definição qual é o símbolo inicial da gramática. Em SSL, o símbolo inicial é definido apenas implicitamente e o mecanismo de definição de seções poderia ser usado para obscurecer essa função. A segunda alteração se deve à liberdade de composição de nomes de seção, que usa a notação "`<>`". O uso dos símbolos `<` e `>` em contextos tão diferentes poderia tornar uma descrição confusa, assim, preferimos eliminar essa possível perda de legibilidade.

A linguagem SDL também sofreu alterações. Como em SSL, a notação de especificação de tuplas "`<...>`" foi eliminada, permanecendo "`(...)`". SDL permite a decoração de identificadores denotando variáveis com dígitos e apóstrofos como forma de especificação de uma "família de variáveis" [Big81]. A decoração também é válida em identificadores ligados a domínios, porém, não há nenhum tipo de família de domínios. Para evitar analogias entre o uso das formas de decoração, em LDS, identificadores de domínios não podem ser decorados.

Como em *literate programming*, a chave para obtenção de boas definições semânticas é o equilíbrio entre exposição formal e informal. A parte informal de cada seção deve ser usada tanto para explicar os aspectos tratados na parte formal quanto para explicar o próprio formalismo usado. Essa interação entre equações e documentação deve servir de ajuda na escolha da

melhor forma de estruturar a descrição. A definição e uso de trechos de código (equações) deve ser considerada tendo em vista a estrutura final da descrição. LDS não restringe a forma que a parte formal pode assumir, deixando o autor livre para buscar a codificação mais adequada segundo seus critérios.

LDS procura oferecer uma estrutura que unifique texto formal e informal. O objetivo não é apenas a apresentação simultânea das duas formas de definição, mas também sua produção simultânea, como resultado, melhores definições podem ser obtidas. Apenas um pequeno número de modificações foram realizadas nas linguagens e poucos comandos próprios de LDS foram inseridos, assim, LDS pode manter razoavelmente simples a interação das linguagens que a compõem e oferecer ao usuário amplo controle sobre as definições produzidas.

4 O Uso de LDS

A aplicação de LDS está associada aos usos de semântica denotacional. Assim, podemos esperar o uso de LDS principalmente na mecanização da produção de compiladores, como ferramenta na formulação de provas de correção de programas e no projeto de novas linguagens de programação.

Sistemas tradicionais de geração de compiladores usam especificações que incluem trechos de código. Semântica denotacional dirige o enfoque de descrições de linguagens de programação para conceitos abstratos de seu significado, permitindo uma expressão independente de implementação. Ainda assim, é possível a geração automática de compiladores a partir de descrições denotacionais. A geração automática de compiladores permite a rápida obtenção de uma implementação "correta" de uma linguagem, i.e., uma implementação de acordo com sua implementação formal.

O emprego de sistemas de geração de compiladores pode trazer os seguintes benefícios: 1) maior confiabilidade na correção de definições formais, pois o sistema poderia realizar verificações de consistência da definição, e o autor teria uma visão operacional para confrontar com a interpretação semântica tencionada; 2) projetistas de linguagens poderiam avaliar decisões e testar alternativas de projeto a partir de bases teóricas (a definição formal) e práticas, pois programas de teste na linguagem projeta poderiam ser formulados e executados através do compilador gerado; 3) os conceitos precisos e princípios de projeto de semântica denotacional seriam mais acessíveis a projetistas e programadores, pois uma interpretação em termos de implementação (o compilador gerado) estaria disponível; 4) implementadores de linguagens poderiam usar uma implementação padrão para avaliação de compiladores codificados manualmente. Embora os compiladores produzidos automáticamente possam não ser eficientes, sua correção pode ser estabelecida através da definição formal.

Semântica denotacional fornece o ambiente, os mecanismos e a infomação semântica necessárias a uma abordagem matemática de manipulação de programas. As propriedades de programas podem ser formuladas numa forma abstrata, e provas de correção podem ser construidas. O rigor formal de descrições denotacionais torna seu uso interessante em outras aplicações associadas à implementação de linguagens de programação. Essas aplicações estão geralmente ligadas à geração automática de compiladores, e incluem, entre outras, análise de fluxo de dados para otimização de programas [Cha88], e a transformação de programas seqüenciais em programas paralelos [Jou87].

O projeto de linguagens de programação envolve questões subjetivas como intuição, estilo, preferências e experiências com outras linguagens [Kli85]. Métodos de descrição de linguagens de programação podem ser usados como ferramentas de projeto de linguagens, contribuindo com a base teórica e a formalização para análise e orientação nas decisões de projeto. A teoria matemática presente faz de semântica denotacional a abordagem de definição semântica mais indicada ao uso em projeto de linguagens, pois um entendimento mais profundo do processo de programação pode contribuir para a formalização dos príncipios de projeto.

[Lig75] propõe o uso de proposições ao estilo de semântica axiomática para a declaração de propriedades das construções de uma linguagem. Um conjunto de hipóteses de funcionamento e uma definição denotacional da linguagem orientada à validação dessas proposições formam a base para a formulação e teste das proposições. Um critério de projeto estabelece que as construções devem ter propriedades definidas através de um pequeno número de hipóteses, cujas provas são fáceis de formular, i.e., sua definição denotacional é simples. O objetivo do critério é trazer maior claridade semântica, através de propriedades mais simples e fáceis de compreender.

[Ten77] identifica dois princípios baseados em semântica denotacional que podem ser aplicados ao projeto de linguagens. O princípio da correspondência estabelece a necessidade do projeto conjunto de mecanismos similares de uma linguagem, em especial mecanismos declarativos e paramétricos. Irregularidades e restrições desnecessárias seriam evitadas, resultando em linguagens mais simples e uniformes. O princípio da abstração sugere a identificação das estruturas com valor semântico e a criação dos mecanismos de abstração correspondentes. O conjunto de abstrações da linguagem seria mais coerente e completo, e a linguagem mais poderosa.

LDS está inserido num ambiente de definição de semântica denotacional [Ama92] que procura oferecer ao seu usuário ferramentas para diminuição do esforço de composição de descrições de linguagens de programação. O objetivo principal do ambiente é o aumento na qualidade das descrições; essa melhora de qualidade significa definições mais confiáveis e fáceis de ler e compreender. O ambiente fornece uma série de serviços orientados à formulação e teste de descrições denotacionais através de uma interface homem-máquina de alto nível. A lista dos serviços disponíveis se compõe de edição de definições e programas de

teste, interface com a linguagem de formatação LaTeX, verificação de tipos nas definições, geração automática de compiladores e execução de programas de teste.

As funções oferecidas pelo ambiente de definição semântica procuram fornecer liberdade (no ambiente e na linguagem de definição) como principal meio para obtenção de melhor documentação de linguagens de programação. Porém, as possibilidades de utilização do ambiente se estendem ainda ao seu uso como ferramenta de projeto e implementação de linguagens, através de princípios de projeto baseados em semântica denotacional e oferecendo uma implementação do compilador da linguagem para teste com programas.

5 Conclusões

Semântica denotacional apresenta muitas virtudes como método de especificação de linguagens de programação, mas tem deficiências de legibilidade. A linguagem de especificação LDS foi projetada com o objetivo de atenuar os problemas de legibilidade de semântica denotacional.

Uma das formas de aumentar a legibilidade de descrições semânticas é a incorporação de modularidade. A estrutura imposta pelos mecanismos de modularização tornam as descrições mais adequadas à apresentação. LDS deve sua maior legibilidade em parte ao uso da linguagem modular de descrição semântica SDL. Técnicas de *literate programming* foram utilizadas em LDS; dessa forma, um modo informal de exposição foi incorporado à linguagem. O modo informal traz uma estrutura de documentos a LDS, i.e., LDS possui mecanismos de divisão em hierarquias de seções para melhor apresentação das definições. Cada seção de uma definição em LDS apresenta um aspecto da linguagem descrita. Os modos de exposição se completam e reforçam em cada seção, resultando em definições legíeis e precises, e assim , em maior facilidade de compreensão e maior confiança nas definições.

Referências

[Ama92] L.M. Amaral. Interface homem-máquina do ambiente de definição de semântica LDS. Master's thesis, Universidade Federal de Minas Gerais, Belo Horizonte – MG, 1992.

[AO90] A. Avenarius and S. Opperman. `FWEB`: A literate programming system for FORTRAN 8x. SIGPLAN Notices, 25(1), janeiro 1990.

[BGW78] R. Balzer, N. Goldman, and D. Wille. Informality in program specifications. IEEE Transactions on Software Engineering, SE-4(2), març 1978.

[Big81] R.S. Bigonha. A Denotational Semantics Implementation System. PhD thesis, University of California, Los Angeles, Cal., 1981.

[Big84] R.S. Bigonha. A methodology for structuring denotational definitions. Série de Monografias do Departamento de Ciência da Computação T01/84, Universidade Federal de Minas Gerais, Belo Horizonte – MG, 1984.

[BJ75] F.P. Brooks Jr. The Mythical Man-Month. Addison-Wesley, Reading, Mass., 1975.

[BLAR92] R.S. Bigonha, J. Leite S. Jnr, L.M. Amaral, and W.A. Rodrigues. Semântica denotacional legível. Seminário informática 25, Pontifícia Universidade Católica, Rio de Janeiro – RJ, agosto 1992.

[CB91] D. Cordes and M. Brown. Literate-programming paradigm. IEEE Computer, 24(6), junho 1991.

[Cha88] S.J. Chao. Denotational semantics for program analysis. SIGPLAN Notices, 23(1), janeiro 1988.

[Den87] P.J. Denning. Annoucing literate programming. Communications of the ACM, 30(7), julho 1987.

[Jou87] P. Jouvelot. Semantic parallelization: a practical exercise in abstract interpretation. In Fourteenth Symposium on Principles of Programming Languages, pages 29–38, Munich, janeiro 1987. ACM SIGACT-SIGPLAN.

[Kli85] P. Klint. A study in string Processing Languages, Lecture Notes in Computes Science 205. Springer-Verlag, Berlin, 1985.

[Knu84] D.E. Knuth. Literate programming. The Computer Journal, 27(2), maio 1984.

[Knu83] D.E. Knuth. The `web` system of structured documentation. Technical Report STAN-CS-83-980, Stanford University, Stanford, Cal., setembro 1983.

[Knu86] D.E. Knuth. The TEXbook, volume A of Computers and Typesetting. Addison-Wesley, Reading, Mass., 1986.

[Lam86] L. Lamport. LATEX: A Document Preparation System. Addison-Wesley, Reading, Mass., 1986.

[Lei93] J. Leite S. Jnr. Linguagem de Definição e Geração de Analisadores Sintáticos em Semântica Denotacional Legível. Master's thesis, Universidade Federal de Minas Gerais, Belo Horizonte – MG, 1993.

[Lig75] G.T. Ligler. A mathematical approach to language design. In Second Symposium on Principles of Programming Languages, Palo Alto, Cal., 1975. ACM.

[Mos78] P.D. Mosses. SIS — A Compiler-Generator System Using Denotational Semantics. Daimi, University of Aarhus, 1978.

[PJN85] A.J.H.M. Peels, N.J. Jansen, and W. Nawijn. Document architecture and text formatting. ACM Transactions on Office Information Systems, 3(4), outubro 1985.

[Pag81] A style for writing the syntatic portions of complete definitions of programming languages. The Computer Journal, 24(2), maio 1981.

[Rod93] W.A. Rodrigues. Compilação e Otimização de uma Linguagem para Definição Semântica Denotacional. Master's thesis, Universidade Federal de Minas Gerais, Belo Horizonte – MG, 1993.

[RS89] T.Reenskaug and A.L. Skaar, An environment for literate smalltalk programming. In OOPSLA '89 proceedings, pages 337-345, Sidney, outubro 1989.

[Sol86] E. Soloway. Learn to program = learning to construct mechanisms and explications. Communications of the ACM, 29(9), setembro 1986.

[SS71] D. Scott and C. Stratchey. Towards a mathematical semantics for computer languages. Technical Monograph PRG-6, Oxford University Computing Laboratory, agosto 1971.

[Ten77] R.D. Tennent. Language design methods based on semantic principles. Acta Informatica, 8(2), 1977.

[Thi86] H. Thimbleby. Experiences of 'literate programming' using cweb (a variant of Knuth's `WEB`. The Computer Journal, 29(1), março 1986.

[vWHG87] C.J. van Wyk, D.R. Hanson, and J. Gilbert. Literate programming — printing common words. Communications of the ACM, 33(3), março 1990.

[vW90] C.J. van Wyk. Literate programming: an assessment. Communications of the ACM, 33(3), março 1990.

[WB89] Y.C. Wu and T.P. Baker, A source code documentation for Ada. ACM Ada Letters, 9(5), julho/agosto 1989.

[Wei84] M. Weiser. Program slicing. IEEE transactions on Software Engineering, SE-10(4), julho 1984.