

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 alternativa à descrição 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.

Como resultado foi definida a linguagem de especificação LDS. LDS une prosa informal e equações semânticas numa mesma estrutura, promovendo um estilo de descrição onde a ênfase se encontra na estruturação e apresentação das definições.

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

Abstract

This work presents a new alternative to the description of programming languages. One assumption underlies this alternative: the main problems of formal definitions are their low legibility and difficulty of comprehension. To solve these problems, it was verified the suitability of applying the techniques of *literate programming* to denotational semantics.

The result was the design of the specification language LDS. This language associates in the same structure informal prose and semantic equations, promoting a style of definition composition where the emphasis is in the definition structuring and presentation.

[keywords]: programming languages, formal descriptions, denotational semantics, 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.

Métodos informais de definição carregam consigo uma imprecisão semântica que pode resultar em descrições muitos extensas, inconsistentes ou ambíguas. Demonstrando que seu uso é inadequado para a 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 [2].

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.

Propomos a associação numa estrutura única do formalismo presente em semântica denotacional à naturalidade da prosa informal. Esse estilo foi inspirado em *literate programming*, onde um programa é formado por seções compostas de código e texto informal. O equilíbrio entre texto formal e informal resultaria em descrições que atingem 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 [4] 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, sendo precisas e consistentes, e ainda assim legíveis e naturais.

2 Programação Legível

O termo usado no título dessa seção procura capturar a essência do título original: *literate programming*. O fundamental em *literate programming* é a 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.

Em *literate programming*, programas são tratados como obras literárias. 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.

Em *literate programming* programas são divididos em seções. Cada seção apresenta algum aspecto da solução representada pelo programa, não havendo restrições quanto a ordenação das seções. As seções podem incluir prosa e código, ou apenas um deles, o objetivo é expor cada característica do programa 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. As seções representam os componentes mais simples de um programa e a estrutura de apresentação demonstra o relacionamento entre esses componentes.

Na mesma linguagem há uma linguagem de formatação, para uma melhor apresentação a pessoas, e uma linguagem de programação, 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. *Weave* procura satisfazer as necessidades de legibilidade das pessoas. Seu objetivo é realçar a estrutura dos pro-

gramas. A formatação da documentação e código é considerada para uma melhor apresentação. Todo o aparato técnico de índices, sumários e referências cruzadas é gerado.

2.1 Problemas com *literate programming*

Um dos primeiros problemas é o caráter multilíngüe de *literate programming*. A notação é uma combinação de uma linguagem de formatação, uma linguagem de programação e comandos próprios da nova notação. Programadores devem se sentir confortáveis e preparados para lidar com essa profusão de notações. O aprendizado da também pode ser mais demorado.

Um problema sutil é originado pelo uso de formatação para produzir a documentação de programas: textos bem formatados fazem o programador buscar maior embelezamento do seu programa, deixando questões funcionais e de estrutura sem a devida atenção. A manutenção de programas pode ser prejudicada. Programadores podem relutar em corrigir documentos bem formatados, pois "um texto formatado parece muito bom para estar errado" [16].

O esquema de macro-expansão (das seções de código) pode trazer dois problemas. O primeiro é a aparente atomicidade do uso do código, pois há liberdade sintática na composição, mas o uso das seções está restrito à sintaxe da linguagem. As soluções do problema envolvem análise sintática das seções (sem restringir a composição de seções). O segundo problema é a difusão de erros pelo uso repetido de uma seção com erro. A ausência de localidade nas seções é a origem do problema. A incorporação de localidade pode ser examinada como solução.

O enfoque excessivo no caráter literário de programas pode causar inconvenientes. Textos literários apresentam material bem conhecido e estável. Programas, geralmente, sofrem contínua modificação e extensão. Novas versões rapidamente tornam as antigas obsoletas; assim, a apresentação não deve receber cuidados excessivos em momentos do desenvolvimento onde a possibilidade de alterações seja grande. Se a metáfora literária for usada criteriosamente não haverá problemas, 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 ser considerado uma desvantagem, mas a qualidade da documentação produzida contrabalança esses custos através da maior facilidade de compreensão de programas.

2.2 Vantagens de *literate programming*

A filosofia de *literate programming* indica um maior realce na necessidade de legibilidade de programas, destacando a importância da comunicação da intenção dos programadores. A audiência humana de um programa passa a receber maior atenção. 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.

O programador não escreve apenas um programa comentado, mas um artigo sobre o programa e seu processo de desenvolvimento. *Literate programming* garante que o artigo pode ser executado. Documentação e código ganham uma maior intimidade eliminando ambigüidades e expondo mais claramente seu relacionamento. Há uma melhora qualitativa na documentação, devido ao maior destaque e o relacionamento mais claro com o código.

Programas são produzidos a partir de pequenos segmentos de código e documentação. Cada segmento representando uma abstração. 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 da linguagem, o programador pode buscar melhor estilo e riqueza de estrutura.

Finalmente, devemos salientar a flexibilidade de *literate programming*, tanto pela sua aplicabilidade a um amplo conjunto de linguagens (e.g., Pascal [6], C [16] e Smalltalk [12]) como pela sua liberdade de composição e leitura de programas. Como observado em [17], "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.

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).

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 agregação de um modo informal de exposição. LDS é o resultado da união de uma linguagem de formatação e duas linguagens de definição formais (uma linguagem de especificação sintática e uma de especificação semântica). O projeto de LDS está baseado em considerações sobre o sistema de documentação WEB [7] e os detalhes de seu projeto estão em [9].

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. A definição pode ser apresentada numa hierarquia de capítulos, seções, subseções, etc. mantendo a referência a textos literários, enquanto sugere a estrutura presente. 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 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. 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. Analisamos a possibilidade de restringir

o código de funções, procedimentos ou módulos iniciados numa seção deverem ser terminados naquela seção. Assim, as dependências entre as seções da definição seriam mais simples e cada seção poderia ser vista como uma unidade funcional completa. Porém, essa restrição iria de encontro a uma das bases de *literate programming*: a liberdade 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 incorpora 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. O escritor tem certo 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 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 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, extraíndo 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 em [9].

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 [10], 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 " [8]. \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 arranjo 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 .

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.

[8] 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. \LaTeX também 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 à 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. [8] 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 [11] 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. A sintaxe abstrata desempenha um papel fundamental em definições denotacionais, como observado em [3]: "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 pode ser vista como uma implementação da sintaxe abstrata da linguagem, assim, a correspondência entre a sintaxe abstrata e a concreta deve ser formalizada em definições completas de linguagens. O objetivo é 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.

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 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 o uso da linguagem de especificação semântica. SSL se encaixa num sistema para descrição completa (sintaxe e semântica) de linguagens de programação, assim, não parecem haver problemas 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 diminuído 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.

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 concreta 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). [3] 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 [14] 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 [3]. 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 [3]. Módulos internos não são presentes em LDS, então não mais consideraremos essa característica de SDL.

[3] 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 todas as questões relacionadas a tipos de entidades sejam resolvidas 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. Como extensão à notação padrão, SDL oferece um mecanismo de modularização que torna as definições semânticas mais legíveis e fáceis de formular e depurar. A descrição completa de SDL e sua definição formal se encontram em [3]. [13] 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-lingüístico 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 $\backslash\text{typein}$ e $\backslash\text{typeout}$ 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 ($\backslash\backslash$, $\backslash\text{newline}$, $\backslash\text{linebreak}$ e $\backslash\text{nolinebreak}$) e nova página ($\backslash\text{clearpage}$, $\backslash\text{cleardoublepage}$, $\backslash\text{pagebreak}$, $\backslash\text{nopagebreak}$, $\backslash\text{samepage}$ e $\backslash\text{newpage}$) 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 $\&\text{start}$; 2) a notação de tuplas " $\langle \dots \rangle$ " foi eliminada. Em SSL, há duas formas de especificação de tuplas: " $\langle \dots \rangle$ " e " (\dots) ", 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 " $\langle \dots \rangle$ ". O uso dos símbolos \langle e \rangle 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 " $\langle \dots \rangle$ " foi eliminada, permanecendo " (\dots) ". SDL permite a decoração de identificadores denotando variáveis com dígitos e apóstrofos como forma de especificação de uma "família de variáveis" [3]. 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 Ambiente de Definição e Implementação de Linguagens de Programação

LDS está inserida num ambiente de definição de semântica denotacional cujo objetivo é oferecer ao seu usuário ferramentas para diminuição do esforço de composição de descrições de linguagens de programação. O ambiente procura 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.

A interface do ambiente de definição semântica fornece os recursos para edição de textos, manipulação de módulos de descrições denotacionais e seleção de arquivos. As operações realizadas estão fundamentadas em três princípios de projeto: facilidade de operação, relevância de informação (apenas as informações necessárias estão presentes em diálogos com o usuário), e resposta semântica (verificação da entrada e das informações apresentadas ao usuário). A arquitetura do ambiente, a interface homem-máquina e seu projeto são discutidas e [1].

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 da implementação automática do compilador da linguagem para testes com programas.

5 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 projetada 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 automaticamente 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 construídas. 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, e a transformação de programas seqüenciais em programas paralelos.

O projeto de linguagens de programação envolve questões subjetivas como intuição, estilo, preferências e experiências com outras linguagens. 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 princípios de projeto.

[15] 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.

6 Um Exemplo

Nessa seção apresentamos um pequeno exemplo de descrição semântica em LDS. A descrição é basicamente a mesma da página 90 de [5], e está presente apenas para dar uma visão das descrições produzidas através do ambiente descrito na seção anterior.

Aqui descrevemos a construção "jumpout". A expressão "jumpout" Id in Exp é uma construção de desvio em expressões. Seu resultado é obtido da seguinte forma:

- A expressão Exp é "executada". Se durante sua execução for encontrada a aplicação da função Id a um argumento Exp' , a execução de Exp é interrompida e o valor de "jumpout" será o valor da avaliação da sub-expressão Exp' .
- Se não houver aplicação da função Id , o valor de "jumpout" será o valor da expressão Exp .

Por exemplo, a expressão:

```
"jumpout" Esc "in" (( "if" Boolean "then" Esc(1) "else" 2) + 2).
```

É equivalente a

```
"if" Boolean "then" 1 "else" (2 + 2).
```

Formalmente, a semântica da expressão "jumpout" é descrita no módulo `Sem-exp` abaixo. Esse módulo importa continuações, um ambiente e o domínio de identificadores, que são definidos adequadamente em outros módulos.

```

MODULE Sem-exp
  DEFINE
    evaluate := Exp → Env → Eval-cont → Exec-cont; ! Semântica de expressões
  DEFINE
    Exp = ["jumpout" Id "in" Exp]
          | (Outras formas de expressão);
    (Cláusulas de importação)
  END Sem-exp

BODY Sem-exp
  DEF
    evaluate(exp) env exp-cont =
      CASE exp
      ▷ ["jumpout" id "in" exp'] →
        evaluate (exp') env {(λ exp1 exp-cont1. exp-cont exp1) ▷ id} exp-cont
      ▷ (Equações semânticas das outras formas de expressão)
  END
END Sem-exp

```

æ

7 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 legíveis e precisas, e assim, em maior facilidade de compreensão e maior confiança nas definições.

Referências

- [1] 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.
- [2] R. Balzer, N. Goldman, and D. Wille. Informality in program specifications. IEEE Transactions on Software Engineering, SE-4(2), março 1978.

- [3] R.S. Bigonha. A Denotational Semantics Implementation System. PhD thesis, University of California, Los Angeles, Cal., 1981.
- [4] F.P. Brooks Jr. The Mythical Man-Month. Addison-Wesley, Reading, Mass., 1975.
- [5] M.C.J. Gordon. The Denotational Description of Programming Languages. Springer-Verlag, New York, N.Y., 1979.
- [6] D.E. Knuth. Literate programming. The Computer Journal, 27(2), maio 1984.
- [7] D.E. Knuth. The web system of structured documentation. Technical Report STAN-CS-83-980, Stanford University, Stanford, Cal., setembro 1983.
- [8] L. Lamport. \LaTeX : A Document Preparation System. Addison-Wesley, Reading, Mass., 1986.
- [9] 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.
- [10] 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.
- [11] F.G. Pagan. A style for writing the syntatic portions of complete definitions of programming languages. The Computer Journal, 24(2), maio 1981.
- [12] T. Reenskaug and A.L. Skaar, An environment for literate smalltalk programming. In OOPSLA '89 proceedings, pages 337-345, Sidney, outubro 1989.
- [13] 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.
- [14] D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. Technical Monograph PRG-6, Oxford University Computing Laboratory, agosto 1971.
- [15] R.D. Tennent. Language design methods based on semantic principles. Acta Informatica, 8(2), 1977.
- [16] H. Thimbleby. Experiences of 'literate programming' using cweb (a variant of Knuth's WEB). The Computer Journal, 29(1), março 1986.
- [17] C.J. van Wyk, D.R. Hanson, and J. Gilbert. Literate programming — printing common words. Communications of the ACM, 33(3), março 1990.

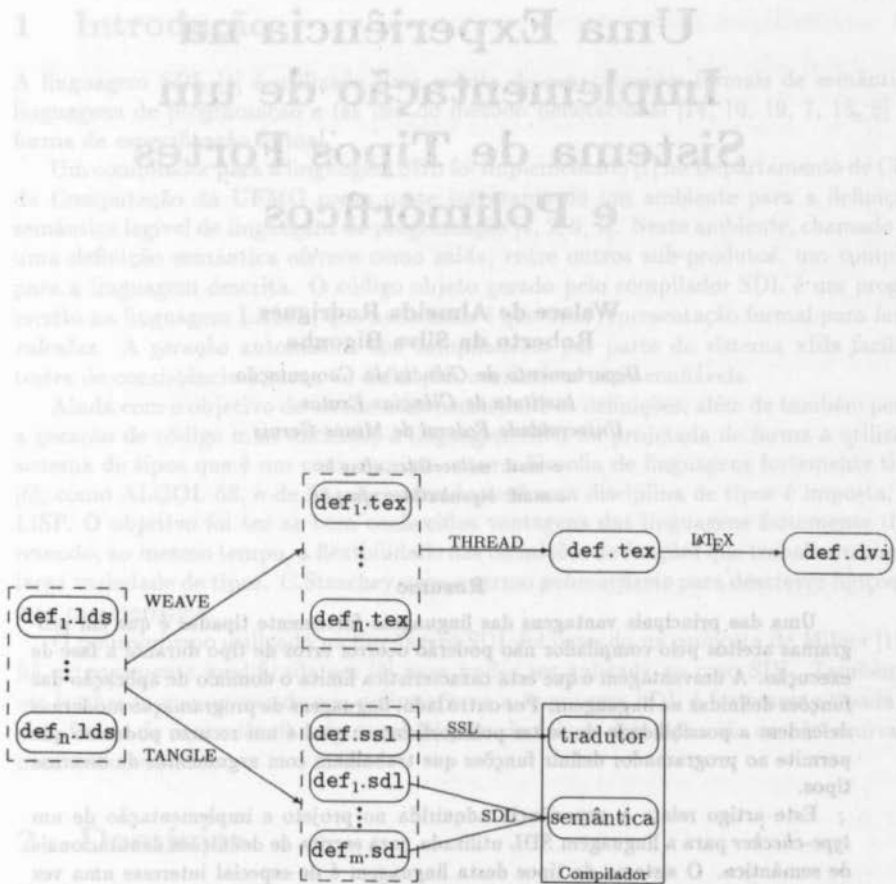


Figura 1: WEB semântico

This paper reports the experience in the design and implementation of a type checker for the declarative semantics language SDL, which contains a strong type discipline with polymorphism.