TAYS CRISTINA DO AMARAL PALES SOARES

COMPILAÇÃO DE SEMÂNTICA DENOTACIONAL MODULAR

Belo Horizonte, Minas Gerais
Abril de 2007

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

COMPILAÇÃO DE SEMÂNTICA DENOTACIONAL MODULAR

Dissertação apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

TAYS CRISTINA DO AMARAL PALES SOARES

Belo Horizonte, Minas Gerais
Abril de 2007

UNIVERSIDADE FEDERAL DE MINAS GERAIS

FOLHA DE APROVAÇÃO

Compilação de Semântica Denotacional Modular

TAYS CRISTINA DO AMARAL PALES SOARES

Dissertação defendida e aprovada pela banca examinadora constituída por:

Roberto da Silva Bigonha – Orientador
Universidade Federal de Minas Gerais

Mariza Andrade da Silva Bigonha – Co-orientador
Universidade Federal de Minas Gerais

Belo Horizonte, Minas Gerais, Abril de 2007

Resumo

A semântica e a sintaxe de linguagens de programação podem ser descritas de forma precisa por meio de definições formais. Semântica Denotacional é uma abordagem que permite a descrição formal de linguagens de programação. Essa abordagem apresenta, entretanto, problemas de modularidade. De fato, a especificação formal de linguagens de grande porte é inerentemente complexa, surgindo a necessidade de um modelo capaz de decompor uma descrição em módulos de maneira que a complexidade possa ser controlada. Uma nova metodologia, denominada Semântica Multidimensional, permite que definições em semântica denotacional sejam escritas modularmente. A linguagem funcional de domínio específico *Notus* segue o modelo proposto por esta nova abordagem.

Este trabalho apresenta a implementação de um compilador para a linguagem *Notus*, que permite a escrita incremental de especificações léxica e sintática de linguagens, usando gramáticas livres de contexto, e da especificação semântica, em semântica denotacional. A modularidade é guiada pela sintaxe abstrata da linguagem que se deseja definir. Para cada construção sintática semanticamente relevante, define-se um módulo. Isto facilita a obtenção de reusabilidade e extensibilidade em especificações em semântica denotacional. O compilador de *Notus* gera interpretadores para linguagens especificadas. Apresentam-se ainda, neste texto, os problemas decorrentes da compilação incremental de definições modulares, e as soluções para esses problemas implementadas no compilador *Notus*. O compilador desenvolvido foi testado com especificações modulares em semântica denotacional para linguagens de teste, escritas incrementalmente.

Abstract

Programming language semantics and syntax can be defined without ambiguities using formal definitions. Denotational Semantics is an approach which allows formal description of language semantics. However, uses of Denotational Semantics face scalability issues. Formal specification of large scale languages is inherently complex, requiring a model capable of decomposing semantics descriptions in such a way that it is possible to manage its complexity. Multidimensional Semantics proposes a new methodology for development of modular denotational semantic definitions. The functional, domain specific, language Notus implements the model proposed by this new approach.

This work presents a compiler implementation for Notus, which is a language that allows writing syntactic, lexical and semantic definitions incrementally. The modularity is based on the desired language's abstract syntax. A module is made for each semantically relevant syntactic construct. This improves the reusability and extensibility in denotational semantics specifications. The Notus compiler produces interpreters for specified languages. Compilation of modular specification poses issues that are also solved in the Notus compiler.

Agradecimentos

Agradeço aos meus pais, Helenice e René, o suporte e carinho; ao Eduardo, a dedicação, companheirismo e paciência; aos professores Roberto e Mariza Bigonha o apoio, a confiança, e as inúmeras correções neste texto; aos amigos do LLP o ambiente de trabalho descontraído, em especial, ao Fabio Tirelo o suporte no desenvolvimento deste trabalho.

Sumário

Capítulo 1

Introdução

Apesar dos esforços já realizados no desenvolvimento de uma diversidade de abordagens para descrições formais da semântica das linguagens, o seu uso na especificação de linguagens de programação de grande porte é, na prática, ainda pequeno
[Zhang e Xu, 2004, Mosses, 2001].

Mosses enumera os usos potenciais e reais de uma descrição semântica [Mosses, 2001]. A semântica formal pode ser usada para registrar decisões de projeto realizadas durante a criação de linguagens ou como forma de documentação de referência. Conceitos usados nas linguagens podem ser compreendidos no estudo de especificações formais, assim como se pode obter novas percepções sobre esses conceitos.

Compiladores, interpretadores e protótipos podem ser gerados a partir de descrições formais, permitindo a avaliação e o raciocínio sobre propriedades de programas escritos na linguagem especificada. Entretanto, os compiladores e interpretadores assim gerados não são eficientes a ponto de serem usados na prática. Na verdade, o uso mais comum desse modelo é a geração de protótipos para verificação da correção da própria descrição.

Em contraste com o uso restrito de descrições semânticas, as descrições formais da sintaxe de linguagens de programação são amplamente utilizadas. Mosses atribui esse fato à sua facilidade de escrita e compreensão. Os principais conceitos como alternativas, seqüenciamento e recursão podem ser representados de maneira uniforme e existem ferramentas que suportam prototipação de gramáticas e geração de analisadores sintáticos eficientes, como por exemplo o Yacc [Brown et al., 1992].

Segundo Mosses [Mosses, 2001], o uso ainda pequeno de descrições formais da semântica de linguagens decorre principalmente da dificuldade para se escrever e ler definições usando as propostas existentes, e do reduzido número de ferramentas e ambientes com o amparo necessário para a geração de protótipos, interpretadores e compiladores que auxiliem na validação de descrições semânticas formais.

A dificuldade de se especificar a semântica de linguagens de programação de grande porte é inerente à sua complexidade. Para que seja possível controlar a complexidade é desejável que uma descrição formal possa ser decomposta em pequenos módulos independentes, e que possa ser feita de forma incremental a partir da composição desses módulos.

O objetivo deste trabalho é o desenvolvimento de um ambiente para descrições em semântica denotacional de linguagens de programação de forma modular, buscando facilitar a especificação de linguagens de programação de grande porte. As descrições léxica, sintática e semântica de linguagens de programação são realizadas, no ambiente proposto, usando a linguagem *Notus*, especificada como parte da proposta de tese de doutorado de Tirelo [Tirelo, 2005]. A ferramenta desenvolvida gera protótipos na forma de interpretadores para as linguagens especificadas.

Não é objetivo essencial deste trabalho que os interpretadores gerados sejam eficientes. No entanto, esforços são feitos para que o ambiente proposto gere interpretadores que apresentem tempos de execução aceitáveis para uso prático.

A proposta da linguagem *Notus*, no início desse trabalho, era permitir a escrita incremental de uma descrição semântica, eliminando a possível interferência decorrente da definição de um novo módulo em partes já escritas por meio de mônadas e transformadores de mônadas. Adicionalmente, princípios da orientação por aspectos seriam usados no auxílio à escrita modular de características de natureza transversal das linguagens. No entanto, o compilador desenvolvido foi construído em paralelo ao projeto da linguagem, e mudanças no seu projeto substituíram essas construções da linguagem. O compilador *Notus* foi preparado para receber as novas construções, descritas na Seção 3.8, cuja implementação constitui-se em um trabalho futuro de extensão do compilador desenvolvido.

O foco deste trabalho é, portanto, a construção de um compilador de especificações modulares em semântica denotacional, no qual o objetivo de escrita incremental de especificações foi atingido para as partes léxica e sintática de linguagens. A parte semântica é escrita de forma tradicional, como em [Gordon, 1979]. A modularidade das descrições realizadas no ambiente proposto é alcançada por meio de recursos lingüísticos para definição de pacotes, módulos, controle de visibilidade, e construções para extensão das descrições léxica e sintática.

A modularidade é guiada pela sintaxe abstrata da linguagem. Por exemplo, normalmente escreve-se em *Notus* um módulo para a especificação da semântica de comandos da linguagem. Nesse módulo são descritos os componentes léxico, sintático e semântico referentes aos seus comandos. Realiza-se o mesmo processo para o módulo de expressões e outros elementos da sintaxe abstrata da linguagem.

No compilador desenvolvido, os módulos que compõem a descrição formal da lingua-

gem são transformados em analisadores léxico, sintático e semântico para a linguagem especificada, que agrupados formam um interpretador. Dificuldades na compilação de especificações modulares surgem a partir da união dos módulos para geração de código. Os problemas são relacionados à correta ordenação dos componentes léxico, sintático e semântico durante o processo de união para a geração de analisadores únicos da sintaxe e da semântica da linguagem especificada. Como essas questões são transparentes para programadores da linguagem *Notus*, elas devem ser resolvidas pelo compilador antes da geração de um interpretador para a especificação.

As contribuições deste trabalho são:

- criação de um compilador para a linguagem *Notus*, que produz interpretadores a partir de linguagens especificadas em semântica denotacional, permitindo prototipar linguagens de programação;

- aplicação de técnicas e algoritmos para geração de analisadores léxico e sintático a partir de definições modulares;

- determinação da ordem de geração das macros de forma que toda macro seja declarada antes de seu uso;

- determinação da ordem de geração das expressões regulares que definem conjuntos não disjuntos de *tokens* da linguagem, de forma a decidir sobre ambiguidades;

- tratamento semântico uniforme entre as produções definidas no módulo da declaração de uma variável de gramática e as regras definidas em módulos diferentes;

- tradução do sistema de tipos e subtipos de *Notus* para *datatypes* em *Haskell* via injeção e projeção de valores de *datatypes*.

- tratamento uniforme para definições de uma mesma função com números variados de parâmetros explícitos;

- identificação de ambiguidades entre os domínios semânticos da especificação;

- desenvolvimento de técnicas de compilação para manter a correspondência na tradução das definições de funções semânticas realizadas de forma modular em *Notus*, para a forma como *Haskell* as processa;

- tradução de funções *Notus* com domínio primitivo para funções *Haskell* sem *overhead* de compilação.

Apesar de a metodologia proposta em [Tirelo, 2005] não ter sido completamente implementada por ainda estar em desenvolvimento, o compilador apresentado neste trabalho poderá ser ampliado com as novas construções da linguagem, quando definidas.

O ambiente proposto para especificação de semântica formal modular e geração de interpretadores para linguagens é constituído por um compilador, responsável pelo reconhecimento da linguagem especificada e pela geração de um interpretador em *Haskell* para essa linguagem.

A Figura 1.1 mostra em alto nível a geração de um interpretador a partir dos módulos da definição de uma linguagem L em *Notus*[1], e o processo pelo qual um programa em L é executado a partir de sua descrição formal em *Notus*; o interpretador gerado recebe um programa escrito em L e as entradas para esse programa e gera a saída.

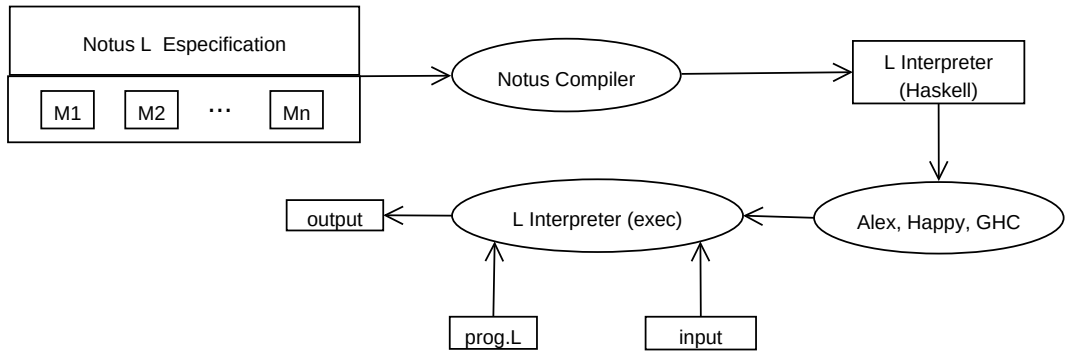

Figura 1.1: Geração de um interpretador para linguagem L a partir de uma especificação de L em *Notus*

1.1 Organização Deste Texto

O Capítulo 2 apresenta uma visão geral das principais abordagens para definições formais da semântica de linguagens de programação. Problemas de modularidade nessas abordagens são discutidas na seção de conclusões desse capítulo. A linguagem funcional de domínio específico *Notus*, uma proposta de solução para os problemas de modularidade de semântica denotacional, é apresentada no Capítulo 3. Os Capítulos 4 e 5 apresentam o compilador da linguagem *Notus*, bem como o processo de compilação, descrevendo os problemas decorrentes da tradução de descrições modulares e as soluções implementadas no compilador *Notus* para esses problemas. O compilador *Notus* e interpretadores gerados a partir de especificações modulares são avaliados no Capítulo 6. Os resultados obtidos e trabalhos futuros são resumidos no Capítulo 7.

[1]Os módulos são identificados na figura por M_1, M_2, ..., M_n

Capítulo 2

Modularidade em Semântica Formal

A semântica e a sintaxe de linguagens de programação podem ser descritas de forma precisa por meio de definições formais. A sintaxe refere-se à forma das expressões que são permitidas pela linguagem, já as definições semânticas podem descrever os efeitos da execução de uma expressão sintaticamente correta ou um programa, ou ainda podem descrever como são executadas. A descrição da sintaxe livre de contexto das linguagens é comumente feita com gramáticas livre de contexto por meio da *Forma de Backus-Naur*, que fornece meios adequados para especificar e raciocinar sobre a gramática de uma linguagem. Em contraste, para as descrições formais da semântica das linguagens existe uma diversidade de abordagens. Dentre elas destacam-se a Semântica Axiomática [Hoare, 1969], a Semântica Operacional [Plotkin, 1981, Gurevich, 1995] e a Semântica Denotacional [Stoy, 1977]. Esta diversidade se deve principalmente ao fato de que o comportamento de um programa é naturalmente mais complexo que sua estrutura [Zhang e Xu, 2004].

Técnicas formais usadas para descrição da semântica de linguagens de programação fornecem conceitos independentes de máquina, técnicas de especificação sem ambigüidades e base teórica rigorosa que suportem raciocínio confiável para provas e deduções [Gordon, 1979]. A semântica formal é também aplicada como guia de validação de implementação e base para geração automática de compiladores [Zhang e Xu, 2004].

As seções seguintes apresentam uma visão geral das principais abordagens para definições formais da semântica de linguagens de programação. A Seção 2.2.1 descreve a Semântica Axiomática, a Seção 2.2.2 descreve a Semântica Operacional, e a Seção 2.2.3 descreve a Semântica Denotacional. Abordagens propostas que têm por objetivo resolver problemas em aberto nas abordagens supracitadas também serão descritas na Seção 2.3. É importante destacar que as diferentes abordagens existentes para

descrições formais da semântica de linguagens não são concorrentes entre si, mas se complementam, cada uma com suas vantagens e desvantagens.

2.1 Propriedades Desejáveis da Especificação Formal

Segundo [Moura, 1996, Zhang e Xu, 2004], um método de especificação formal de linguagens de programação deve apresentar legibilidade, modularidade, capacidade de abstração, comparação, raciocínio, aplicabilidade e ferramentas de suporte.

Legibilidade é importante para facilitar a percepção de propriedades inerentes à linguagem de programação especificada por meio de uma simples análise de sua descrição, tornando a descrição acessível a todas as pessoas interessadas na linguagem.

A modularidade está relacionada a reuso e facilidade de modificação de descrições formais, provendo meios para que módulos usados em uma especificação sejam reaproveitados em descrições de outras linguagens, e que descrições possam ser modificadas alterando-se apenas partes pontuais. Descrições de linguagens de programação reais são geralmente extensas, e a modularidade possibilita a decomposição da especificação em pequenos componentes manipuláveis independentemente, promovendo controle sobre sua complexidade.

A abstração presente no formalismo da descrição deve ser suficiente para que os projetistas da linguagem sejam capazes de se concentrar na especificação, sem ter que se ater aos detalhes de implementação.

A descrição formal de linguagens de programação deve permitir comparações entre linguagens. Para que isso seja alcançado, é preciso que o método de descrição apresente um certo padrão a ser seguido pelas descrições.

O objetivo do formalismo é facilitar a compreensão de programas escritos na linguagem especificada, apresentando-se como uma forma não ambígua, e portanto clara, de descrever aspectos específicos da linguagem de programação. A descrição deve permitir raciocínio sobre programas escritos na linguagem.

O método formal deve ser capaz de descrever conceitos presentes nas linguagens de programação como estado, entrada e saída (I/O), exceções e não-determinismo, sendo assim aplicável da forma mais direta possível às linguagens de programação reais.

Ferramentas de suporte são importantes para auxiliar na escrita, verificação e leitura das descrições semânticas. O uso prático e real das descrições semânticas depende da disponibilidade de ferramentas capazes de gerar interpretadores e compiladores a partir das descrições formais.

Todas as propriedades descritas são importantes, no entanto, a ausência de ferramentas de suporte a uma especificação pode impedir que linguagens de programação sejam nela descritas, mesmo se o arcabouço utilizado na especificação possuir as demais características descritas nesta seção, pois a validação da descrição e seu uso prático estariam comprometidos.

2.2 Semântica de Linguagens de Programação

2.2.1 Semântica Axiomática

A Semântica Axiomática envolve asserções sobre valores de variáveis, regras de inferência e axiomas [Mosses e Watt, 1986]. Uma definição axiomática define regras de inferência por meio de axiomas para construções atômicas da linguagem, em conjunto com teoremas. A semântica da linguagem baseia-se em métodos de dedução lógica de cálculo de predicados de primeira ordem, e é comumente usada para prova de correção de programas, por meio da análise estática de seu código fonte [Slonneger, 1995, Capítulo 11]. Os principais problemas dessa abordagem estão relacionados à generalidade e facilidade de compreensão. Por exemplo, o comando **goto** é difícil de ser especificado, e existem problemas com regras de escopo e procedimentos com parâmetros, além de expressões não poderem possuir efeito colateral.

A semântica do programa é descrita pela enumeração de asserções que devem ser válidas antes e após a execução de um comando; essas asserções têm a forma:

$$\{PRE\}C\{POST\},$$

onde o comando C está correto se este é executado com valores que fazem PRE (asserção de pré-condição) ser válida, e ao fim da execução de C os valores resultantes fazem $POST$ (asserção de pós-condição) ser válida.

As regras de inferência possuem a forma mostrada na Figura 2.1. Se $H_1, H_2, ..., H_n$ podem ser verificadas, então pode-se concluir que H é válida.

$$\frac{H_1, H_2, ..., H_n}{H}$$

Figura 2.1: Forma da regra de inferência

O comando **if-then-else** pode ser definido como mostra o código da Figura 2.2

$$\frac{\{PRE\ and\ B\}\ C_1\ \{POST\}\quad ,\ \{PRE\ and\ (not\ B)\}\ C_2\ \{POST\}}{\{PRE\}\ \textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{endif}\ \{POST\}}$$

Figura 2.2: Exemplo de especificação em Semântica Axiomática para o comando **if-then-else**

A semântica do comando ***if*** envolve uma escolha entre duas alternativas, ou C_1 é executado ou C_2 é executado. Assim duas asserções devem ser satisfeitas para uma dada expressão booleana B, para que o comando *if* seja válido.

2.2.2 Semântica Operacional

Em contraste com as demais técnicas de especificação semântica que descrevem ***o que*** um programa faz, a Semântica Operacional descreve ***como*** a execução de um programa é realizada [Slonneger, 1995, Capítulo 8]. Descrições operacionais completas de linguagens de programação reais podem ser documentos volumosos, por abordarem detalhes de natureza de implementação, normalmente desnecessários para a compreensão da semântica da linguagem [Mosses e Watt, 1986]. A semântica Operacional Estruturada (SOS) descreve como os resultados de cada etapa da execução são obtidos. A Semântica Operacional de uma linguagem de programação é especificada em termos da execução do programa em máquinas abstratas [Slonneger, 1995, Seção 8.4], onde as definições são feitas por meio de um sistema de regras de inferência, consistindo de uma conclusão alcançada mediante um conjunto de premissas e com possíveis condições. A forma geral das regras de inferência é:

$$\frac{premissa_1 \quad premissa_2 \ \cdots \ premissa_n}{\text{conclusão}} \quad \text{condição}$$

Uma regra de inferência sem premissa é chamada de axioma. O comando **if-then-else**, em semântica operacional estruturada, pode ser definido como mostra o código da Figura 2.3.

$$\frac{\langle E, s\rangle \rightarrow \langle E', s'\rangle}{\langle if\ E\ then\ C_1\ else\ C_2,\ s\rangle \rightarrow \langle if\ E'\ then\ C_1\ else\ C_2,\ s'\rangle}$$

$$\langle if\ true\ then\ C_1\ else\ C_2,\ s\rangle \rightarrow \langle C_1, s\rangle$$

$$\langle if\ false\ then\ C_1\ else\ C_2,\ s\rangle \rightarrow \langle C_2, s\rangle$$

Figura 2.3: Exemplo de especificação em SOS para o comando **if-then-else**

Em [Mosses, 2005], Mosses apresenta uma proposta de modularização para SOS. Segundo Mosses, nessa abordagem, denominada MSOS, as regras de transição são feitas de forma incremental e a adição de uma nova construção na linguagem especificada não implica na alteração das construções já definidas, como ocorre em SOS convencional.

2.2.3 Semântica Denotacional

Em Semântica Denotacional, a semântica de uma linguagem de programação é descrita em termos de objetos matemáticos. O termo denotacional refere-se ao fato que os construtos da linguagem são descritos por denotações, entidades matemáticas abstratas que modelam o significado de cada elemento sintático da linguagem [Gordon, 1979]. A Semântica Denotacional é composicional, ou seja, a semântica de cada construto é uma função da semântica de seus constituintes.

Uma definição tradicional em Semântica Denotacional é constituída pelas seções: domínio sintático, sintaxe abstrata, domínio semântico, funções semânticas e equações semânticas. No domínio sintático são listadas as categorias sintáticas da linguagem especificada. A sintaxe abstrata especifica os construtos pertinentes a cada categoria sintática. Domínios semânticos determinam os objetos matemáticos que compõem a semântica da linguagem. Funções semânticas mapeiam objetos do mundo sintático em objetos do mundo semântico. A seção de funções semânticas exibe as assinaturas das funções, e a seção de equações semânticas especifica a denotação semântica de cada construto sintático da linguagem. A Figura 2.4, extraída do Capítulo 2 do Livro do Gordon [Gordon, 1979], exemplifica o uso da semântica denotacional na especificação formal de uma pequena linguagem.

2.3 Modularidade em Semântica Denotacional

Semântica Denotacional de linguagens de programação define a semântica de cada construto da linguagem e sua contribuição para a resposta final do programa. Em definições modulares escritas de forma incremental, algumas informações de contexto, como *environments* e *stores*, só são identificadas quando necessárias. A não antecipação de tal problema pode ocasionar a redefinição de módulos existentes.

A deficiência da semântica denotacional em relação a modularidade pode ser percebida pelos exemplos da Figura 2.5 extraídos de [Liang, 1998]. Na Figura 2.5(a) é mostrada a semântica denotacional de uma linguagem aritmética simples. A função que representa a denotação de expressões dessa linguagem mapeia termos (*term*) em valores (*value*). A modularidade de uma descrição pode ser mensurada pelas alterações necessárias quando se inclui uma nova característica na linguagem. Na Figura 2.5(b),

Domínios Sintáticos:
$Ide \quad = \quad \{I | I \text{ é um identificador}\}$
$Exp \quad = \quad \{E | E \text{ é uma expressão}\}$
$Com \quad = \quad \{C | C \text{ é um comando}\}$

Sintaxe Abstrata:
$C \quad ::= \quad I := E$
$\quad\quad | \quad\quad \textbf{\textit{while}}\ E\ \textbf{\textit{do}}\ C$
$\quad\quad | \quad\quad C_1; C_2$
$E \quad ::= \quad 0 \mid I \mid \textbf{\textit{suc}}\ E$

Domínios Semânticos:
$\boldsymbol{Num} = \{0, 1, 2, \cdots\}$
$\boldsymbol{State} = [\boldsymbol{Ide} \mapsto \boldsymbol{Num}]$
$n \in \boldsymbol{Num}, s \in \boldsymbol{State}$

Funções Semânticas:
$\mathcal{E} : \boldsymbol{Exp} \mapsto \boldsymbol{State} \mapsto \boldsymbol{Num}$
$\mathcal{C} : \boldsymbol{Com} \mapsto \boldsymbol{State} \mapsto \boldsymbol{State}$

Equações Semânticas:
$\mathcal{E}[\![0]\!]s = 0$
$\mathcal{E}[\![I]\!]s = s\ I$
$\mathcal{E}[\![\textbf{\textit{suc}}\ E]\!]s = \mathcal{E}[\![E]\!]s + 1$
$\mathcal{C}[\![I \ := \ E]\!]s = s[\mathcal{E}[\![E]\!]s/I]$
$\mathcal{C}[\![C_1; C_2]\!]s = \mathcal{C}[\![C_2]\!](\mathcal{C}[\![C_1]\!]s)$
$\mathcal{C}[\![\textbf{\textit{while}}\ E\ \textbf{\textit{do}}\ C]\!]s = \mathcal{E}[\![E]\!] = 0 \rightarrow s, \mathcal{C}[\![\textbf{\textit{while}}\ E\ \textbf{\textit{do}}\ C]\!](\mathcal{C}[\![C]\!]s)$

Figura 2.4: Exemplo de especificação em Semântica Denotacional

é inserido um *environment* que mapeia nomes de variáveis a valores, de forma que a linguagem suporte variáveis. Nota-se que a denotação para números passa a lidar com um *environment*, apesar de não depender dele. A operação aritmética '+' também não depende diretamente do *environment*, no entanto sua denotação precisa ser alterada para se adequar à nova funcionalidade.

Ao se incluírem seqüenciadores na linguagem, pode ser necessário migrar para semântica de continuação, cujo impacto nas definições é mostrado na Figura 2.5(c). A inclusão dessa nova funcionalidade também implica em mudanças nas descrições já existentes. De forma geral, este exemplo ilustra a necessidade de mudanças globais nas descrições em semântica denotacional tradicional, ao se acrescentarem novas funcionalidades nas linguagens de programação descritas. Essa "fragilidade" da semântica denotacional motiva o estudo de novas abordagens que visam alcançar a modularidade necessária para a descrição de linguagens de programação reais [Liang, 1998, Mosses, 1988]. Algumas dessas propostas são descritas nas seções a seguir.

$$\mathcal{E} : Term \rightarrow Value$$
$$\mathcal{E}[\![n]\!] = n$$
$$\mathcal{E}[\![e_1 \ + \ e_2]\!] = \mathcal{E}[\![e_1]\!] + \mathcal{E}[\![e_2]\!]$$

(a) Definição em Semântica Denotacional de uma linguagem aritmética simples

$$\mathcal{E} : Term \rightarrow Env \rightarrow Value$$
$$\mathcal{E}[\![n]\!] = \lambda r.n$$
$$\mathcal{E}[\![e_1 \ + \ e_2]\!] = \lambda r.\mathcal{E}[\![e_1]\!]r + \mathcal{E}[\![e_2]\!]r$$
$$\mathcal{E}[\![v]\!] = \lambda r.r \ v$$

(b) Inclusão variáveis à definição em semântica denotacional

$$\mathcal{E} : Term \rightarrow Env \rightarrow Cont \rightarrow Ans$$
$$\mathcal{E}[\![n]\!] = \lambda r \ k.kn$$
$$\mathcal{E}[\![e_1 \ + \ e_2]\!] = \lambda r \ k.\mathcal{E}[\![e_1]\!]r(\lambda i.\mathcal{E}[\![e_2]\!]r(\lambda j.k(i+j)))$$
$$\mathcal{E}[\![e_1; e_2]\!] = \lambda r \ k.\mathcal{E}[\![e_1]\!]r(\lambda x.\mathcal{E}[\![e_2]\!]rk)$$

(c) Inclusão de continuação para suportar seqüenciadores

Figura 2.5: A falta de modularidade em semântica denotacional

2.3.1 Semântica de Ações

Semântica de Ações (AS[1]) [Mosses, 1988, Mosses, 1996a, Moura, 1996][Mosses, 1996b, Mosses e Watt, 1986] é um método para especificação formal de linguagens de programação, originalmente desenvolvido por Peter Mosses, em 1977, com contribuições de David Watt a partir de 1984. Semântica de Ações é um arcabouço (*framework*) que mistura técnicas das semânticas denotacional e operacional em conjunto com leis algébricas, e com uma notação própria (*action notation*). A Semântica Denotacional é o ponto de partida para o desenvolvimento de AS, destacando-se a utilização de gramáticas livres de contexto para a representação da sintaxe abstrata das linguagens, e de equações semânticas para prover definições intuitivas de funções semânticas composicionais. Essas funções semânticas mapeiam os elementos sintáticos em elementos semânticos.

Semântica de Ações é um método de descrição formal que apresenta as características definidas na Seção 2.1 [Moura, 1996, Mosses e Watt, 1986]. Sua notação é verbosa e sugestiva, tendo por objetivo ser acessível a programadores. A semântica da linguagem é descrita por um conjunto padrão de *ações* primitivas e combinadores de *ações*, que proporcionam reuso de partes de uma descrição já feita em uma nova descrição, facilitando a comparação entre linguagens a partir de suas descrições semânticas.

[1] do inglês *Action Semantics*

A Semântica de Ações é composicional, assim como a Semântica Denotacional [Gordon, 1979], ou seja, a semântica de um construto da linguagem é função da semântica de seu constituintes. No entanto, a semântica não é representada por funções de ordem superior, mas por "ações"que, segundo seus autores, são mais simples por possuírem interpretação operacional. O principal objetivo da Semântica de Ações é prover meios de uma descrição formal se aproximar, em termos de facilidade de leitura e compreensão, de uma descrição em linguagem natural, mas com as vantagens inerentes ao formalismo, tais como precisão e base teórica para provas.

Na descrição de uma linguagem de programação usando Semântica de Ações, observam-se *ações* representando computações. A *ação* correspondente a um construto é composta de *ações* primitivas e combinadores de *ações*. Conceitualmente, uma *ação* é uma entidade que pode ser avaliada para processar informação [Mosses, 1988]. Em geral, o resultado da avaliação de uma *ação* pode ser uma terminação normal (*completion*), terminação com exceção (*escape*), não-terminação (*divergence*) ou falha (*failure*).

As informações processadas pelas ações são classificadas como temporárias (*transients*), com escopo (*scoped*), estáveis (*stable*) e permanentes (*permanent*). As informações temporárias são tuplas de dados correspondendo a resultados intermediários para uso imediato na *ação*. As informações com escopo são ligações (*bindings*) de dados a *tokens*, correspondendo à tabela de símbolos. As informações permanentes são os dados comunicados entre ações distribuídas.

As ações possuem facetas[2] distintas, que se referem à propagação de informação. As facetas podem ser básicas, funcionais, imperativas, declarativas e comunicativas. Na faceta básica ocorre o processamento independentemente da informação, corresponde ao fluxo de controle; na faceta funcional, *ações* recebem e produzem conjuntos de valores nomeáveis, correspondentes a informações temporárias, ou seja, dados passados de uma *ação* para a outra; na faceta imperativa, *ações* recebem e podem alterar conteúdos de posições de armazenamento, ou seja, manipulam informações estáveis; na faceta declarativa, ações recebem e produzem conjuntos de ligações de *tokens* a valores em um determinado escopo; a faceta comunicativa atua sobre informação permanente recebendo e enviando mensagens. A importância dessas facetas está relacionada à notação padrão de *ações*. Cada *ação* primitiva atua em uma determinada faceta e não produz ou modifica informações pertencentes a outras facetas.

Uma descrição em AS é composta por três módulos: Sintaxe Abstrata, Entidades Semânticas e Funções Semânticas. No primeiro módulo é feita a especificação da sintaxe abstrata da linguagem por meio de uma gramática livre de contexto semelhante à notação *BNF*. São apresentados um conjunto de não-terminais e suas respectivas

[2]No inglês, o termo usado é *facets*

produções. Alguns símbolos não-terminais, como identificadores e numerais, não precisam ser especificados.

O segundo módulo especifica todas as entidades semânticas a serem usadas nas funções semânticas. Essa especificação é dada por meio da listagem de conjuntos (*sorts*) e operadores. Entidades semânticas padrão, bem como ações primitivas, combinadores de ações e alguns tipos, são fornecidos pela notação de *ação* padrão. Nesse módulo definem-se os tipos dos dados e os operadores que serão usados nas funções semânticas, tais como a determinação de valores primitivos e dados armazenáveis. Essa definição consiste em especializar as entidades semânticas padrão.

Existem três tipos de entidades semânticas: ações, dados e *yielders*. Ações consistem em entidades computacionais cujas execuções representam o comportamento de processamento de informação e refletem a natureza gradual, passo-a-passo, da computação. Itens de dados são entidades matemáticas de natureza estática, que representam pedaços de informação e que são processados pelas ações. Como exemplos de dados comuns têm-se os valores booleanos, números, caracteres, *strings* e listas. Um *yielder* representa um item de dado não avaliado e seu valor depende das informações correntes, como valores presentes na entrada. Durante a execução de uma *ação*, um *yielder* é avaliado, gerando um dado.

No terceiro módulo são definidas as funções semânticas que mapeiam a sintaxe da linguagem em sua semântica. Para cada construto da linguagem é escrita uma equação semântica que usa constantes e operações definidas pelas entidades semânticas. Uma função semântica possui como entrada um único elemento sintático e produz como saída uma entidade semântica.

Para seus autores, Semântica de Ações se apresenta como um método bastante atraente para especificações formais de linguagens de programação. Segundo [Mosses e Watt, 1986], AS tenta encontrar um caminho intermediário entre o rigor do formalismo e as armadilhas das descrições informais, buscando reconciliação entre a teoria e a prática. Em [Mosses, 1988], Peter Mosses afirma que AS apresenta boa modularidade devido à sua notação padrão denominada *action notation*, mas observa que sua notação verbal não está relacionada a modularidade, mas à facilidade de leitura e compreensão.

Os criadores de AS afirmam que não existem garantias de que os combinadores de ações existentes possam expressar todas as maneiras possíveis de combinar ações e que talvez seja necessário incluir novos combinadores [Mosses e Watt, 1986]. Esse fato torna-se algo preocupante ao ser defrontado com o que se encontra em [Mosses, 1996b], onde Mosses afirma que mudanças na semântica operacional da notação de *ação* ou a criação de novos combinadores de ações implicariam em uma reformulação de todas as leis da notação de *ação*, e que a notação de *ação* existente é difícil de ser alterada.

Em [Mosses, 1996a] são citados alguns exemplos e ferramentas que permitem o desenvolvimento de definições formais usando Semântica de Ações tais como uma especificação formal de Pascal disponível em [Mosses e Watt, 1993] e ferramentas como ASD [van Deursen e Mosses, 1996] e Abaco System [Moura et al., 2002].

AS permite a escrita de especificações em semântica denotacional com grande facilidade e alto grau de legibilidade. No entanto, este modelo não permite a abstração de determinadas informações de contextos em equações de construtos que não tratam diretamente com essas informações, e dessa maneira, algumas construções da linguagem devem considerar aspectos que não estão diretamente ligados a essas informações. Por exemplo, os comandos de atribuição e condicional não são diretamente dependentes de continuações e, no entanto, na semântica de ações de Pascal, as equações semânticas para esses comandos devem considerar a existência de continuações. Além disso, não é possível modularizar regras para verificação de tipos dinâmica.

2.3.2 Semântica Monádica Modular

A Semântica Monádica Modular (MMS)[3] [Liang e Hudak, 1996, Liang et al., 1995] é uma abordagem monádica para modularizar descrições em semântica denotacional. MMS permite que linguagens de programação de grande porte sejam definidas em pequenos blocos modulares reusáveis, que podem ser combinados de acordo com o conjunto de características de cada linguagem. Para seus autores, as principais contribuições dessa abordagem para a semântica denotacional convencional são as facilidades de especificação, raciocínio e implementação das linguagens de programação.

Na próximas seções são definidos os principais conceitos inerentes à Semântica Monádica: mônadas, unidades de construção (*building blocks*), e transformadores de mônadas.

2.3.2.1 Mônadas

Na Semântica Monádica, mônadas são usadas para capturar individualmente a essência das mais variadas características de linguagens de programação complexas, abstraindo detalhes de baixo nível como *environments* e *stores*. Uma mônada é uma família de tipos $m\ a$ com um construtor de tipo polimórfico m mostrada na Listagem 2.1.

```
class Monad m where
  (>>=)  :: m a -> (a-> m b) -> m b
  return :: a -> m a
  (>>)   :: m a -> m b -> m b
```

[3]Do inglês *Modular Monadic Semantics*

```
  fail    :: String -> m a
```

Listagem 2.1: A classe mônada

A definição da classe **Monad** possui declarações *default* para as funções (>>) e *fail*, por isso, para se declarar uma mônada em *Haskell*, é obrigatório implementar apenas as funções (>>=) e *return*. Devido a esse fato, é comum definir uma mônada como um construtor de tipos M e duas funções polimórficas *return* e *bind*, sendo que a última corresponde ao '(>>=)' definido na Figura 2.1 [Wadler, 1992, Liang et al., 1995].

A Listagem 2.2 define uma mônada trivial exemplificando a instanciação da classe **Monad**.

```
data Id a = Id a

instance Monad Id where
  return x       = Id x
  (>>=) (Id x) f = f x
```

Listagem 2.2: A mônada Id

Informalmente pode-se dizer que a função *return* permite "entrar" na mônada e o operador (>>=) permite operar dentro da mônada seqüenciando ações, sem que isso implique em "sair" da mônada.

Confrontando as definições em semântica denotacional convencional (Figura 2.5(a)) e a semântica monádica (Figura 2.6) para a mesma linguagem nota-se que a função dessa última, que descreve a denotação dos construtos da linguagem, mapeia termos (*term*) em computações (*M Value*), onde *M* é uma mônada (descrita a seguir na Figura 2.8) e *Value* denota o resultado da computação. Os detalhes intrínsecos à computação tais como *environment* e *store* são abstraídos na mônada M por meio de suas funções *return* e *bind*.

$$
\begin{aligned}
&\mathcal{E} : Term \rightarrow M\ Value \\
&\mathcal{E}[\![n]\!] = return\ n \\
&\mathcal{E}[\![e_1\ +\ e_2]\!] = (\mathcal{E}[\![e_1]\!])\ >>= \\
&\qquad\qquad (\lambda v_1.\ (\mathcal{E}[\![e_2]\!])\ >>= \\
&\qquad\qquad\qquad (\lambda v_2.return(v_1\ +\ v_2)))
\end{aligned}
$$

Figura 2.6: Definição em Semântica Monádica de uma linguagem aritmética simples

No exemplo da Figura 2.6 a semântica do numeral n é uma computação que apenas retorna n como resultado. A semântica de $\mathcal{E}[\![e_1\ +\ e_2]\!]$ é uma computação que calcula o valor de $\mathcal{E}[\![e_1]\!]$ e o associa a v_1, calcula o valor de $\mathcal{E}[\![e_2]\!]$ e o associa a v_2, e por fim retorna $v_1 + v_2$.

Para que essa linguagem suporte variáveis é necessário incluir apenas a equação semântica para variáveis, como pode ser visto na Figura 2.7.

$$\begin{aligned}\mathcal{E}[\![v]\!] = \; & do\ r \leftarrow rdEnv \\ & return\ r\ v\end{aligned}$$

Figura 2.7: Equação semântica para variáveis

Nota-se que não é preciso, ao incluir um *environment* na definição semântica da linguagem, alterar qualquer equação já definida; em contraste com o que ocorre em definições denotacionais tradicionais, como mostrado na Figura 2.5(b). Nesta equação, a função *rdEnv* recupera o *environment* corrente, que está encapsulado na mônadas.

Considerando a equação semântica para $\mathcal{E}[\![e_1 \ + \ e_2]\!]$ da Figura 2.6, a Figura 2.8 ilustra a dependência das equações semânticas em relação à instanciação da mônada subjacente à descrição em semântica monádica da linguagem de programação especificada[4]. Dessa maneira, para adicionar uma nova construção à linguagem é preciso apenas alterar a mônada base da descrição semântica da linguagem de programação, e adicionar a equação semântica para a nova construção, sendo que nenhuma alteração nas equações já existentes é necessária.

$$\begin{array}{lcl} type\ M\ a & = & a \\ return\ x & = & x \\ bind\ e\ k & = & k\ e \end{array} \quad \Longrightarrow \quad \begin{array}{l} \mathcal{E} : Term \rightarrow Value \\ \mathcal{E}[\![e_1 + e_2]\!] \ = \mathcal{E}[\![e_1]\!] + \mathcal{E}[\![e_2]\!] \end{array}$$

(a) Mônada trivial

$$\begin{array}{lcl} type\ M\ a & = & Env \rightarrow a \\ return\ x & = & \lambda r.x \\ bind\ e\ k & = & \lambda r.k\ (e\ r)\ r \end{array} \quad \Longrightarrow \quad \begin{array}{l} \mathcal{E} : Term \rightarrow Env \rightarrow Value \\ \mathcal{E}[\![e_1 + e_2]\!] = \lambda r.\mathcal{E}[\![e_1]\!]\ r + \mathcal{E}[\![e_2]\!]\ r \end{array}$$

(b) Mônada com *environment*

Figura 2.8: Comportamento da denotação para $\mathcal{E}[\![e_1 + e_2]\!]$ de acordo com a mônada subjacente

É importante observar que a definição semântica para $\mathcal{E}[\![e_1 \ + \ e_2]\!]$ não se alterou diante da inclusão de variáveis, ou seja, inclusão do *environment*. A relação das equações semânticas das construções já existentes na linguagem com a nova característica acrescida se faz pela nova mônada definida, o que torna transparente as conseqüências provenientes da inclusão dessa nova funcionalidade. Na Figura 2.8(a) é mostrado o significado de $\mathcal{E}[\![e_1 \ + \ e_2]\!]$ substituindo as operações *return* e *bind* da equação da Figura

[4]A Figura 2.8 é baseada na apresentada por Liang em [Liang, 1998]

2.6 de acordo com a definição dessas funções da mônada subjacente, e o mesmo é feito em 2.8(b), onde a mônada base atua na presença de um *environment*.

Unidades de Construção da Semântica Monádica. As unidades de construção definem a semântica monádica de cada construto da linguagem de programação descrita, tais como comando de atribuição, funções, operações aritméticas e continuações. Cada unidade se caracteriza por sua independência em relação às demais. No entanto, elas se baseiam em um conjunto comum de operações núcleo (*kernel*) da linguagem. Por exemplo, o comando de atribuição e as continuações de desvio podem usar o mesmo *store*. A unidade de construção para operações aritméticas é mostrada na Figura 2.6. A seguir serão exemplificadas uma unidade de construção para funções e uma para alocações e atribuições.

- Unidade de construção para funções:

  Uma unidade de construção para funções é mostrada na Figura 2.9. Para simplificar a leitura das equações semânticas, a função de seqüenciamento de mônadas *bind* será substituída por algo similar à notação "*do*" usada em *Haskell* [Thompson, 1999, Capítulo 18]. O tipo *Value* corresponde à união disjunta de valores básicos da linguagem e um tipo para funções, que mapeia computações em computações.

$$
\begin{aligned}
&\mathcal{E} : Term \rightarrow M\ Value\\
&\mathcal{E}[\![v]\!] = \{r \leftarrow rdEnv;\ r\ v\}\\
&\mathcal{E}[\![\lambda v.e]\!] = \{r \leftarrow rdEnv;\ return(\lambda x.inEnv\ r[x/v]\mathcal{E}[\![e]\!]\}\\
&\mathcal{E}[\![e_1\ e_2]\!] = \{f \leftarrow \mathcal{E}[\![e_1]\!]; v \leftarrow \mathcal{E}[\![e_2]\!]; f(return\ v)\}
\end{aligned}
$$

Figura 2.9: Definição em Semântica Monádica para funções

A descrição em semântica denotacional de abstrações e aplicações de funções precisa acessar um *environment* que mapeia variáveis em computações, fazendo-se necessário o uso de duas operações básicas que atuam sobre o *environment*:

```
type Env = Name → M Value
rdEnv    :: M Env
inEnv    :: Env → M Value → M Value
```

A operação *rdEnv* recupera o *environment* atual e a operação *inEnv* executa uma computação em um *environment* dado. A denotação para variáveis consiste em recuperar o *environment* atual e retornar a computação associada à variável nesse *environment*. A semântica para uma abstração é retornar uma função que

espera um argumento e invoca a operação *inEnv* passando o *environment* atual acrescido da associação da variável de abstração ao argumento dessa função e a denotação da expressão da abstração. Dessa maneira, a denotação da expressão será executada no novo *environment*. Para a denotação de aplicação de função é feita a avaliação da primeira expressão, que produz uma função, seguida pela avaliação da segunda expressão, que produz um valor que é passado para a função.

- Unidade de construção para alocações e atribuições:

  Características comuns em linguagens imperativas tais como alocação de memória e atribuições de variáveis podem ser descritas por meio de um local de armazenamento usualmente denominado *store* que mapeia localizações (do tipo *Loc*) em computações. Operações para alocação, leitura e escrita em posições desse *store* são definidas:

```
type Store = Loc → M Value
alloc      :: M Loc
readS      :: Loc → M Value
write      :: (Loc,M Value) → M ()
```

A Figura 2.10 mostra a semântica monádica para alocações e atribuições. A denotação para referenciar uma expressão corresponde a avaliar essa expressão, obter uma nova localização no *store*, escrever o valor obtido pela avaliação da expressão na nova localização e, por último, retornar o local onde o valor da expressão foi armazenado. Para "derreferenciar" uma expressão, deve-se avaliá-la para produzir uma localização que será usada para recuperar o seu valor associado no *store*. Para a atribuição, avalia-se a expressão do lado esquerdo da atribuição, produzindo uma localização, avalia-se a expressão do lado direito produzindo um valor, e associa-se no *store* a localização obtida à computação correspondente ao valor avaliado.

$$\mathcal{E}[\![ref\ e]\!] = \{v \leftarrow \mathcal{E}[\![e]\!]; l \leftarrow alloc; write(l, return\ v); return\ l\}$$
$$\mathcal{E}[\![deref\ e]\!] = \{l \leftarrow \mathcal{E}[\![e]\!]; read\ l\}$$
$$\mathcal{E}[\![e_1 := e_2]\!] = \{l \leftarrow \mathcal{E}[\![e_1]\!]; v \leftarrow \mathcal{E}[\![e_2]\!]; write(l, return\ v)\}$$

Figura 2.10: Definição em Semântica Monádica para alocações e atribuições

As operações *alloc*, *readS* e *write* abordam uma noção de estado e podem ser definidas em termos da função:

```
update :: (Store → Store) → M Store
```

para um dado tipo *Store*. Para recuperar o estado atual deve-se passar a função identidade para *update*, e para alterar o estado passa-se uma função de transformador de estado. Uma possível definição das funções *alloc*, *readS* e *write* é:

```
type Store = Loc → M Value

alloc       :: M Loc
alloc        = {s ← update (λi.i); return f s}

readS       :: Loc → M Value
readS l      = {s ← update (λi.i); s l}

write       :: (Loc,M Value) → M ()
write (l,v) = {_ ← update (λs.s[v/l]); return ()}
```

O *underscore* ('_') indica que o retorno da função *update* é ignorado. A função f:Store → Loc, usada em *alloc* percorre o *store s* procurando por uma posição de memória que esteja disponível.

Transformadores de Mônadas. Diante dos exemplos das Figuras 2.6, 2.9 e 2.10 nota-se que as unidades de construção referentes a cada característica da linguagem de programação especificada necessitam, além das operações de *bind* e *return* da mônada subjacente, de um conjunto base de operações, como mostra a Tabela 2.1.

Característica	Função
Environment	rdEnv :: M Env inEnv :: Env → M Value → M Value
Store	alloc :: M Loc readS :: Loc → M Value write :: (Loc,M Value) → M ()

Tabela 2.1: Operações monádicas usadas nas definições semânticas

Em definições de semântica denotacional tradicional, ao se listarem as operações auxiliares necessárias às definições semânticas, o passo seguinte é enumerar os domínios e implementar essas funções. No entanto, se futuramente existir a necessidade de incluir na linguagem um novo recurso e novas operações, possivelmente será necessária a redefinição dos domínios semânticos e de equações já construídas.

Transformadores de mônadas permitem que a semântica para cada característica da linguagem seja tratada individualmente e, por meio da operação *lift*, é possível de-

terminar a interação entre as características. Para que a modularidade seja alcançada, a definição semântica começa a partir de uma mônada simples e características vão sendo adicionadas. O papel do transformador de mônada é, dada uma mônada inicial, retornar uma nova mônada que incorpore a característica adicionada.

Formalmente, um transformador de mônada é um construtor de tipos t e uma função associada *lift*, onde t mapeia qualquer mônada $(m, return_m, bind_m)$ para uma nova mônada $(t\ m, return_t\ m, bind_t\ m)$. A Listagem 2.3 exemplifica a criação e o funcionamento de um transformador de mônada. Para elucidar o comportamento dos transformadores de mônadas, é apresentada a seguir a definição dos transformadores de mônadas para *state* e *environment*, seguida da interação entre eles.

Transformador de Mônadas para Estado. De forma semelhante à classe mônada da Listagem 2.1, os transformadores de mônadas, em *Haskell*, pertencem a uma classe de tipos denominada *MonadT* que possui uma função "*lift*" a ser implementada por suas instâncias, mostradas nas Linhas 1 e 2 da Listagem 2.3. Na Linha 4 é criado um tipo de dado que corresponde ao tipo da mônada *state* resultante da aplicação do transformador *StateT s* a uma mônada qualquer *m*. O código das Linhas 7 a 10 completa a declaração da mônada *StateT s m*, criando uma instância da classe mônada de forma que, se *m* é uma mônada, então *StateT s m* também é. Nas Linhas 12 a 14, o transformador de mônadas *StateT s* é declarado uma instância da classe *MonadT* com uma implementação da operação *lift*. Nota-se que a operação *lift* apenas insere a mônada *m* no novo contexto, enquanto o estado é preservado. Por último, uma mônada *StateMonad*, nas Linhas 16 e 17, é definida como uma subclasse de *Monad* com a adição da operação *update*, já que uma mônada para *State* deve suportar essa operação, e nas Linhas 19 e 20 *StateT s* transforma qualquer mônada *m* em uma mônada *state*, onde *update f* aplica *f* ao estado atual.

```
1  class MonadT t where
2          lift :: (Monad m, Monad (t m)) ⇒ m a → t m a
3
4  data StateT s m a = StateM (s -> m (s,a))
5  unStateM (StateM x) = x
6
7  instance Monad m ⇒ Monad (StateT s m) where
8   return x = StateM (\s -> return (s,x))
9   (>>=) (StateM m) k = StateM(\s0->do (s1,a) <- m s0
10                                      unStateM (k a) s1)
11
12 instance MonadT (StateT s) where
```

```
13  lift m = StateM (\s -> do x <- m
14                            return (s,x))
15
16  class Monad m => StateMonad s m where
17   update :: (s->s) -> m s
18
19  instance Monad m => StateMonad s (StateT s m) where
20   update f  = StateM (\s -> return (f s,s))
```

Listagem 2.3: Transformador de mônadas para Estado

Transformador de Mônadas para *Environment*. A Listagem 2.4 ilustra os passos para a criação do transformador de mônadas para *environment*.

```
1   data EnvT r m a = EnvM (r -> m a)
2   unEnvM (EnvM x) = x
3
4   instance Monad m => Monad (EnvT r m) where
5    return x = \r -> return x
6    (>>=) (EnvM m) k = EnvM (\r -> do a <- m r
7                                      unEnvM (k a) r)
8
9   instance MonadT (EnvT r) where
10          lift m = EnvM(\r -> m)
11
12  class Monad m => EnvMonad env m where
13   inEnv :: env -> m a -> m a
14   rdEnv :: m env
15
16  instance Monad m => StateMonad s (StateT s m) where
17   inEnv r m = EnvM(\r' -> m r)
18   rdEnv     = EnvM(\r -> return r)
```

Listagem 2.4: Transformador de mônadas para*Environment*

O transformador de mônadas "EnvT" transforma qualquer mônada m em uma mônada *environment*. As operações disponíveis pelo transformador de *environment* são *inEnv*, que executa uma computação em um *env* dado, ignorando o *env* presente na mônada, e a operação *rdEnv* que apenas insere o *env* dado na mônada transformada.

Interação entre Transformadores de Mônadas. Os transformadores de mônadas inserem as operações necessárias para cada característica suportada pela lingua-

gem. A forma como essas operações se relacionam é dada pelo processo *lifting*, que as transforma por meio do uso da operação *lift* do transformador de mônadas ao qual estão sendo aplicadas. *Lifting* uma operação f para uma mônada m usando-se um transformador t resulta em uma operação em que os tipos $m\ a$ de sua assinatura são substituídos por $t\ m\ a$ como mostra a Figura 2.11.

$$inEnv\ :\ r \to m\ a \to m\ a$$
(a) Assinatura da operação *inEnv* antes da operação de *lifting*

$$inEnv\ :\ r \to t\ m\ a \to t\ m\ a$$
(b) Assinatura da operação *inEnv* depois da operação de *lifting*

Figura 2.11: *Lifting* na operação *inEnv*

As operações *update* e *rdEnv* recebem um tipo não-monádico e retornam uma computação, ou seja, um tipo-monádico. O processo de *lift* para essas funções independe do transformador de mônadas ao qual estão sendo aplicadas, como mostra a Figura 2.12.

$$\begin{aligned} rdEnv_{t\ m} &= lift_t\ .\ rdEnv_m \\ update_{t\ m} &= lift_t\ .\ update_m \end{aligned}$$

Figura 2.12: *Lifting* sobre as operações *rdEnv* e *update*

Já a operação *inEnv* recebe um tipo monádico e por isso o processo de *lifting* dessa operação depende do transformador de mônadas ao qual está sendo aplicada. A Figura 2.13 mostra a operação inEnv sendo aplicada aos transformadores de *environment* *EnvT* e ao de estados *StateT*.

$$\begin{aligned} inEnv_{EnvT\ r'\ m} &: r \to (r' \to m\ a) \to r' \to m\ a \\ inEnv_{EnvT\ r'\ m} r\ e &= \lambda r'.inEnv_m r(e\ r') \\ \\ inEnv_{StateT\ s\ m} &: r \to (s \to m\ (s,a)) \to s \to m\ (s,a) \\ inEnv_{StateT\ s\ m} s\ e &= \lambda s'.inEnv_m s(e\ s') \end{aligned}$$

Figura 2.13: *Lifting* sobre a operação *inEnv* usando os transformadores *EnvT* e *StateT*

De posse dos transformadores de mônadas é possível construir a mônada M subjacente que suporte as unidades de construção para funções, alocações e atribuições. A Figura 2.14 ilustra essa mônada subjacente, onde *Env* e *Store* são tipos de *environment* e *store*.

$$type\ M\ a\ =\ EnvT\ Env\ (StateT\ Store)\ a$$

Figura 2.14: Mônada *M* com: *environment* e *store*

A modularidade em semântica monádica manifesta-se em dois níveis: em alto nível com as unidades de construção e em baixo nível com os transformadores. Usando-se uma série de abstrações, semântica modular monádica transforma a estrutura monolítica da semântica denotacional tradicional em componentes reusáveis [Liang, 1998]. Uma mônada que engloba componentes comuns às definições em semântica denotacional para linguagens de programação, tais como *environment*, continuações, *store* e reportagem de erros pode ser construída como mostra a Figura 2.15.

```
type M a = EnvT Env                 (environment)
             (ContT Answer          (continuação)
                (StateT Store       (store)
                   ErrT)) a         (reportagem de erro)
```

Figura 2.15: Mônada *M* que suporta *environment*, continuações, *store* e reportagem de erros

Um possível problema gerado pela adição de transformadores de mônadas nas definições em semântica monádica é que o número de *liftings* a serem executados pode crescer quadraticamente [Liang et al., 1995]. No entanto, os autores desse arcabouço acreditam que não seja necessário grande número de transformadores.

2.4 Programação Orientada por Aspectos

Modularidade em sistemas significa dividir o *software* em pequenas unidades funcionais, denominadas módulos, que sejam independentes e compiláveis separadamente. No entanto, existem requisitos de um sistema que são transversais a outros, ou seja, cujas implementações se atravessam, dificultando sua definição em apenas uma unidade. Programação Orientada por Aspectos [Kiczales et al., 1997] provê meios de implementar tais requisitos mantendo-se a modularidade geral de sistemas.

A especificação em semântica denotacional de linguagens de programação pode apresentar problemas de modularidade provenientes de entrelaçamento de conceitos presentes nas equações semânticas que as definem. Visando manter alto grau de modularidade, mesmo diante deste fato, a linguagem *Notus* possibilita o uso de aspectos para a implementação de conceitos transversais nas especificações.

A falta de modularidade em sistemas pode ser percebida em manifestações de intrusão e espalhamento. A intrusão ocorre quando o código de mais de um requisito

transversal está presente em uma única região do programa. O espalhamento ocorre quando o código para um determinado requisito transversal encontra-se disperso no sistema. Tirelo apresenta, em [Tirelo, 2005], exemplos de recursos de linguagens de programação, tais como verificação de tipos dinâmicos, tratamento de erros e seqüenciadores, cujas descrições denotacionais apresentam dificuldades de modularização, pois causam intrusão e espalhamento em descrições semânticas.

Um aspecto é normalmente implementado como um componente do sistema que define um requisito transversal, e é aplicado pelo mecanismo de orientação por aspecto em vários pontos da execução de um programa. A união entre o requisito transversal e o resto do sistema é dada pelo processo de costura (*weaving*). O processo de costura consiste em introduzir o código dos aspectos no sistema.

Transformadores de módulos (veja Seção 3.9) são construções em *Notus* inspiradas no processo de costura. Além dessa construção, a estrutura de transmissão de contexto de *Notus* (veja Seção 3.8) é inspirada na implementação de um padrão de projeto, *Wormhole*, implementado em uma linguagem do paradigma de orientação por aspectos, AspectJ [Laddad, 2003], que permite a transmissão implícita das informações de contexto do ponto em que são produzidas para os pontos onde essas são utilizadas.

2.5 Semântica Multidimensional de Linguagens de Programação

Semântica Multidimensional, definida em [Tirelo, 2005], é assim denominada por ser baseada na separação multidimensional de preocupações no desenvolvimento de sistemas e no desenvolvimento de sistemas orientados por aspectos. Essa técnica permite a escrita de definições modulares e extensíveis, de modo que a inclusão de uma nova construção não implique na reescrita de outros módulos já definidos.

Essa metodologia objetiva aperfeiçoar as técnicas de definição de semântica denotacional de linguagens de programação, permitindo modularizar definições de recursos presentes nessas linguagens que se encontram dispersos em uma definição denotacional tradicional.

A decomposição da especificação de uma linguagem é normalmente guiada pelas regras de produção de sua sintaxe abstrata. Semântica multidimensional é baseada na decomposição multidimensional de sistemas e seu principal diferencial é fornecer suporte à decomposição da especificação por meio de, além da sintaxe, outras propriedades da linguagem.

Existem construções em linguagens cuja compreensão não pode ser feita de forma isolada. Em semântica denotacional, essa característica das linguagens prejudica a mo-

dularidade da especificação, já que a descrição de uma construção apresenta elementos de outras, como mostrado na Figura 2.5 da Seção 2.2.3. Com o objetivo de alcançar a modularidade, mesmo diante dessas características das linguagens, semântica multidimensional utiliza construções para a manipulação de contexto e transformação de módulos pelas descrições semânticas de linguagens, possibilitando uma escrita incremental de forma que novos módulos podem ser escritos, iterativamente, sem implicar em alterações nos já existentes.

A principal característica dessa abordagem é permitir o uso de aspectos nas definições da semântica de linguagens, por meio da definição de transformadores de módulos (veja Seção 3.9), de forma a possibilitar a escrita modular de requisitos transversais que causam intrusão e espalhamento em uma descrição, como discutido na Seção 2.4. Transformadores de módulos são inspirados em transformadores de mônadas [Liang et al., 1995] e no processo de costura de código da programação orientada por aspectos [Kiczales et al., 1997]. A diferença principal entre transformadores de mônadas e o de módulos é que o último pode ser seletivamente aplicado a diferente chamadas de funções, e pode ser ignorado quando necessário.

2.6 Conclusões

Os mecanismos para modularidade e extensão de especificações semânticas em abordagens tradicionais, como Semântica Axiomática, Operacional e Denotacional, não são satisfatórios para a especificação de linguagens de programação de grande porte. Propostas com mecanismos mais poderosos para esse tipo de especificação são Semântica de Ações, Semântica Monádica Modular, e a implementada neste trabalho, Semântica Multidimensional.

Semântica de Ações permite a escrita de especificações em semântica denotacional com grande facilidade de escrita e alto grau de legibilidade. No entanto, o projetista deve conhecer inteiramente o conjunto padrão de *ações* primitivas e os combinadores de *ações* existentes. Além disso, como apontado pelos próprios autores desse modelo, a necessidade de definição de um novo combinador de *ação* poderia implicar na reformulação de todas as leis da notação de *ação*. Dessa maneira, apesar desse modelo permitir a escrita incremental de linguagens de programação, o modelo em si não é extensível, o que pode limitá-lo à definição de um conjunto fechado de construções de linguagens de programação.

A Semântica Monádica Modular permite a escrita modular de descrições em semântica denotacional, mas consideramos que a legibilidade e a facilidade de redação são prejudicadas. Entender como os transformadores de mônadas e suas operações de *lift*

operam sobre equações já existentes exige grande esforço do projetista. Além disso, definições escritas nesse modelo são difíceis de ler e conseqüentemente de entender para uma possível extensão e/ou alteração.

Como apresentado na Seção 2.5, definições em semântica denotacional não são extensíveis, uma vez que pequenas mudanças na linguagem podem produzir mudanças espalhadas por toda a definição. *Notus*, uma linguagem funcional de domínio especifico que provê recursos para a especificação modular em semântica denotacional de linguagens de programação, é apresentada neste trabalho. O objetivo da linguagem *Notus* é oferecer uma solução para o grau insatisfatório de modularidade e extensibilidade de especificações em semântica denotacional. *Notus* é a linguagem utilizada para a escrita de definições denotacionais seguindo o modelo de Semântica Multidimensional apresentado na Seção 2.5. O capítulo seguinte apresenta as características e as construções existentes em *Notus* para a escrita de definições modulares em semântica denotacional.

Capítulo 3

A Linguagem de Programação Notus

A linguagem *Notus* provê construções para a escrita incremental modular de especificações em semântica denotacional. *Notus* é uma linguagem puramente funcional com sintaxe semelhante à da linguagem *Haskell*. Este capítulo apresenta as características da linguagem *Notus*, bem como suas construções e os componentes de uma especificação modular em *Notus*.

Notus [Tirelo e Bigonha, 2006] é uma linguagem de domínio específico para a especificação modular de linguagens de programação. *Notus* fornece suporte, na definição de uma linguagem de programação, para:

- divisão de módulos de acordo com as construções;
- especificação da constituição léxica e sintática;
- especificação dos domínios sintáticos e da estrutura da árvore de sintaxe abstrata;
- especificação da semântica separada das construções;
- definição das regras de composição das especificações.

A divisão dos módulos que compõem a descrição formal da linguagem é guiada pela sua sintaxe abstrata. Cada módulo em *Notus* é composto por: (i) importações, onde são enumeradas as dependências do módulo; (ii) especificações léxica e sintática, que definem ou estendem os componentes léxicos e sintáticos de um conjunto de construções; (iii) definições de novos domínios sintáticos e semânticos; (iv) funções semânticas.

A especificação léxica de uma linguagem em *Notus* é dada por meio de expressões regulares semelhantes às utilizadas por geradores de analisadores léxicos, *Lex* [Brown et al., 1992], JLex [Berk, 2003] e Alex [Marlow, 2005a]. A sintaxe é especificada

pelo domínio sintático, utilizando união disjunta e produtos cartesianos para domínios compostos, e pela descrição da gramática da linguagem especificada por um conjunto de regras com notação semelhante à BNF.

Como ferramenta para a descrição formal de linguagens de programação, *Notus* possui como características principais:

- ser uma linguagem puramente funcional *lazy* e apresentar sintaxe semelhante à linguagem *Haskell*;
- permitir organizar em módulos a especificação léxica, sintática e semântica de uma linguagem de programação;
- permitir a escrita de funções de ordem suoerior para a construção das equações que especificam a semântica dos constituintes da linguagem;
- permitir a definição elegante de tipos e subtipos por meio da definição de domínios semânticos;
- permitir a omissão de parâmetros em equações distintas de uma mesma função.

3.1 Pacotes

Os módulos de uma definição em *Notus* podem ser organizados em pacotes, que são conjuntos de módulos. Um módulo `M` pertence a um pacote `P` se:

- o cabeçalho do módulo `M` é `module P.M`;
- o arquivo do módulo `M.nts` está em um diretório de nome `P`.

Pacotes podem conter pacotes, que são chamados de sub-pacotes. Por exemplo, para pacotes aninhados $\texttt{P}_1$, $\texttt{P}_2$, $\cdots$, $\texttt{P}_n$, onde $\texttt{P}_n$ é subpacote de $\texttt{P}_n - 1$, o módulo `M` pertence ao pacote $\texttt{P}_n$ se:

- o cabeçalho do módulo `M` é `module` $\texttt{P}_1\texttt{.P}_2\texttt{.P}_n\texttt{.M}$;
- se existem os diretórios aninhados $\texttt{P}_1$, $\texttt{P}_2$, $\cdots$, $\texttt{P}_n$;
- o arquivo `M.nts` está em um diretório de nome $\texttt{P}_n$.

Os nomes dos pacotes, assim como os nomes dos módulos, possuem a primeira letra em maiúsculo.

Identificadores declarados sem modificador de visibilidade por meio das palavras reservadas `public` ou `private`, possuem visibilidade de pacote, ou seja, são visíveis

pelos módulos do pacote em que foi declarado. Para que um módulo $\texttt{M}_1$ de *Notus* acesse identificadores de outros módulos $\texttt{M}_2$, $\texttt{M}_3$, $\cdots$, $\texttt{M}_n$, o módulo $\texttt{M}_1$ deve importar $\texttt{M}_2$, $\texttt{M}_3$, $\cdots$, $\texttt{M}_n$. Os identificadores visíveis em $\texttt{M}_1$ são os declarados com visibilidade `public` nos módulos importados e os identificadores declarados sem modificador de visibilidade, desde que o módulo de declaração do identificador pertença ao mesmo pacote de $\texttt{M}_1$.

3.2 Elementos do Módulo Principal

O módulo principal de uma especificação em *Notus*, denominado `Main`, define os seguintes elementos:

- função de pré-processamento do arquivo de entrada, f_p;
- símbolo inicial da gramática, `s`, pertencente ao domínio sintático `S`;
- arquivos de entrada, $\langle i_1, i_2, \cdots, i_m \rangle$;
- arquivos de saída, $\langle o_1, o_2, \cdots, o_n \rangle$;
- função semântica de avaliação da *AST*, f_{sem}.

A partir dessa especificação *Notus*, seu compilador gera as funções de análise léxica, (*scanner*) e a função de análise sintática (*parser*). O *scanner* obtém os *tokens* do arquivo fonte pré-processado, e o *parser* produz a árvore de sintaxe abstrata (*AST*) a partir desses *tokens*.

O compilador *Notus* cria também, a partir dos arquivos e funções listados nesta seção, a função principal *main* em *Haskell*, que comanda a execução do interpretador gerado. Assim, toda a especificação é executada na forma de um interpretador que opera sobre um programa fonte fornecido em arquivo de entrada e opcionalmente sobre uma seqüência de entrada, produzindo pelo menos uma seqüência de saída. A execução do interpretador gerado é esquematicamente apresentada no diagrama da Figura 3.1.

O interpretador gerado recebe como primeiro argumento, na linha de comando, o nome do arquivo contendo o programa fonte a ser interpretado, *arq.L*, que é lido e transformado em um *string* chamado *source*, que contém todas as linhas de *arq.L*. O *string source* é passado como argumento para a função de pré-processamento, que retornará um novo *string* $source' = f_p\ source$. Os analisadores léxico e sintático, *Scanner* e *Parser*, atuam sobre o *string* pré-processado $source'$. O *Parser* então produz a *AST* correspondente ao programa fonte. Em seguida, a função semântica f_{sem} recebe como argumentos a *AST* e a seqüência de entradas $\langle i_1, i_2, \cdots, i_m \rangle$, produzindo como resultado a seqüência de saída $\langle o_1, o_2, \cdots, o_n \rangle$. Cada i_k corresponde a um *string* que pode

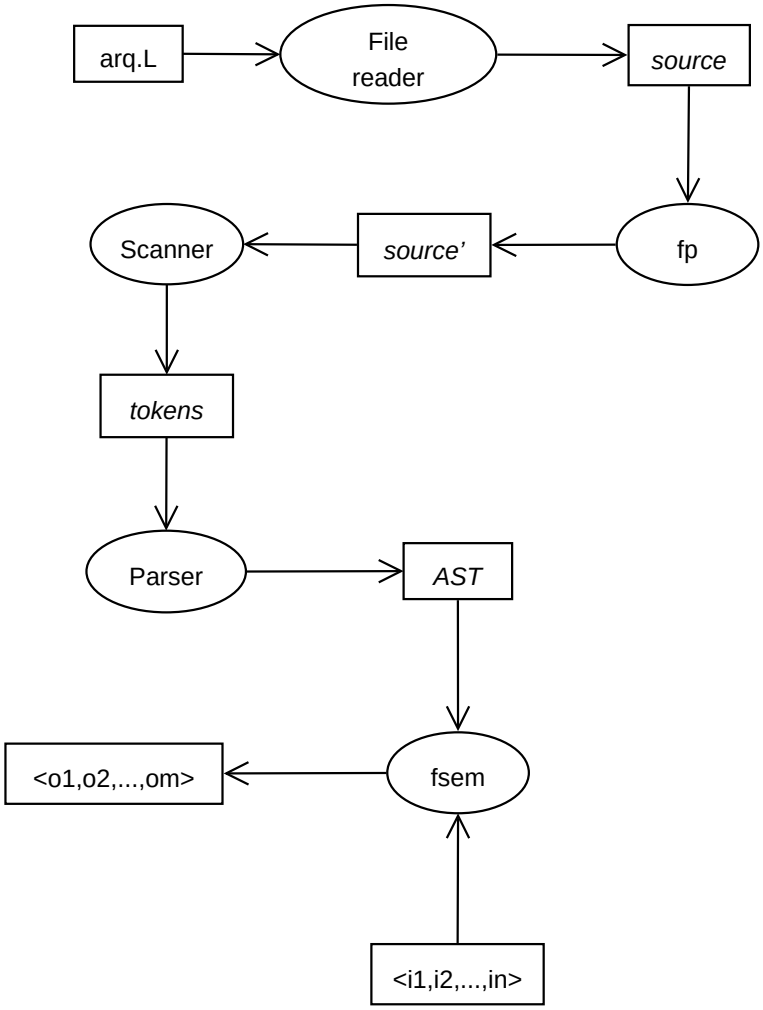

Figura 3.1: Esquema de execução do interpretador gerado para uma linguagem especificada em *Notus*

ser lido da entrada padrão ou de arquivos texto; da mesma forma, cada o_k corresponde a um *string* que será escrito na saída padrão ou em arquivo texto.

Apresenta-se a seguir a declaração dos elementos do módulo principal e os valores *default* de cada um.

3.2.1 Função de Pré-processamento

A função de pré-processamento $\texttt{f}_p$ é definida na especificação da linguagem por meio da cláusula $\langle\texttt{preproc f}_p\rangle$, e pertence ao domínio `String` $\rightarrow$ `String`.

Algumas linguagens, como *Haskell*, definem regras de simplificação de sintaxe, que permitem ao programador a omissão de alguns delimitadores, que podem ser deduzidos pelo contexto. A função de pré-processamento, ao mapear um *string* de entrada em um novo *string*, permite que linguagens com essa característica sejam definidas em *Notus*.

Essa função é opcional, e caso essa não seja definida, o *string* corresponde ao programa de entrada para o interpretador é passado diretamente ao analisador léxico.

3.2.2 Símbolo Inicial da Gramática

O símbolo inicial da gramática é definido por meio da cláusula ⟨`syntax s`⟩, onde `s` é uma variável de gramática concreta pertencente a um domínio `S`. O símbolo inicial determina o domínio da raiz da *AST* produzida pelo analisador sintático. Caso o símbolo inicial da gramática não seja definido, gera-se uma advertência de compilação, e o interpretador resultante não conterá a função `main`.

3.2.3 Seqüências de Entrada

Seqüências de entradas são definidas por meio da cláusula `input in-1, in-2, ... , in-n`, onde cada `in-i` pode ser igual a:

- `stdin`: indica que os dados da seqüência são lidos da entrada padrão;
- um *string* entre aspas duplas: indica que os dados da seqüência são lidos de um arquivo cujo nome é o *string*;
- um identificador `fileId`: indica que os dados da seqüência são lidos de um arquivo a ser associado a `fileId`. A associação é realizada na linha de comando do interpretador, e possui a forma: *fileId=file-name*.

Cada arquivo de entrada deve existir e estar associado a permissões que autorizam a sua leitura. Caso essas condições sejam violadas para qualquer um dos arquivos de entrada, ocorrerá um erro de execução. Por exemplo, a cláusula `input` a seguir:

```
input stdin, "in1.txt", in2;
```

define que o interpretador manipula três seqüências de entrada, a primeira seqüência é lida da entrada padrão, a segunda é lida do arquivo `in1.txt`, e a terceira é lida do arquivo associado ao identificador `in2` na linha de comando do interpretador.

Cada seqüência é manipulada pela função semântica como um *string*. *Notus* disponibiliza funções pré-definidas, listadas na Tabela 3.1 para manipulação das seqüências de entrada.

A cláusula *input* é opcional e, caso o módulo principal não a defina, considera-se que o interpretador não manipula seqüências de entrada ou manipula somente a entrada padrão, de acordo com a assinatura da função semântica (veja Seção 3.2.5).

Função	Domínio	Descrição
`trim`	`String` → `String`	Ignora caracteres em branco no início da seqüência.
`nextInt`	`String` → `Int`	Retorna o próximo inteiro de uma seqüência, ignorando caracteres em branco no início.
`ignoreInt`	`String` → `String`	Ignora caracteres em branco e o primeiro número inteiro no início da seqüência.
`readLine`	`String` → `String`	Retorna todos os caracteres até o primeiro fim de linha da seqüência.
`ignoreLine`	`String` → `Int`	Ignora a primeira linha da seqüência.

Tabela 3.1: Tabela de funções pré-definidas para manipulação de seqüências de entrada

3.2.4 Seqüências de Saída

Seqüências de saída são definidas por meio da cláusula `output out-1, out-2, ... , in-n`, onde cada `out-i` pode ser da forma:

- `stdout`: indica que os dados da seqüência são escritos na saída padrão;
- um *string* entre aspas duplas: indica que os dados da seqüência são escritos em um arquivo cujo nome é o *string*, sobrescrevendo o conteúdo do arquivo caso ele já exista;
- `append fileName`, onde `fileName` é um string entre aspas duplas, indicando que os dados da seqüência são escritos no fim do arquivo de nome `fileName`, caso ele já exista;
- um identificador `fileId`: indica que os dados da seqüência são escritos em um arquivo a ser associado a `fileId`, sobrescrevendo o conteúdo do arquivo, caso ele já exista. A associação é realizada na linha de comando do interpretador, e possui a forma *fileId=file-name*.
- `append fileId`, onde `fileId` é um identificador a ser associado a um arquivo. Os dados da seqüência são escritos no fim do arquivo associado à `fileId`, caso ele já exista. A associação é realizada na linha de comando do interpretador, e possui a forma *fileId=file-name*.

Arquivos de saída não existentes são criados durante a interpretação. Por exemplo, a cláusula `output` a seguir:

```
output stdout, append "out1.txt", out2;
```

define que o interpretador possui três seqüências de saída: a primeira é escrita na saída padrão; a segunda no arquivo de nome `out1.txt`, e a terceira no arquivo associado ao identificador `out2` na linha de comando do interpretador. Caso o arquivo `out1.txt` já exista, a escrita é feita após o seu último caractere; e caso o arquivo associado a `out2` já exista, o seu conteúdo é sobrescrito.

Na ausência da cláusula `output` gera-se uma advertência de compilação, e o intepretador escreve todas as seqüências na saída padrão.

3.2.5 Definição de Função Semântica

A função de avaliação da *AST* é responsável por iniciar a interpretação do programa fonte e é definida por meio da cláusula `semantics f`, onde `f` é uma função com uma das seguintes assinaturas:

1. `S` → `String`: indica que `f` não manipula entrada e produz uma única seqüência de saída;
2. `S` → `String*`: indica que `f` não manipula entrada e produz qualquer número de seqüências de saída;
3. `S` → `String` → `String`: indica que `f` manipula uma única seqüência de entrada e produz uma única seqüência de saída;
4. `S` → `String` → `String*`: indica que `f` manipula uma única seqüência de entrada e produz qualquer número de seqüências de saída;
5. `S` → `String*` → `String`: indica que `f` manipula qualquer número de seqüências de entrada e produz uma única seqüência de saída;
6. `S` → `String*` → `String*`: indica que `f` manipula qualquer número de seqüências de entrada e produz qualquer número de seqüências de saída.

Em todos os casos, o primeiro parâmetro de `f`, S, deve ser o domínio da raiz da *AST*, definida pela cláusula *syntax*, ou seja, `S` é o domínio do símbolo inicial da gramática.

A assinatura da função semântica deve corresponder ao que é definido pelas cláusulas *input* e *output* do módulo principal, sendo que em algumas situações assume-se um comportamento *default*, e em outras geram-se advertências ou erros de compilação ou execução. Por exemplo, se a função semântica pertencer ao caso 1 e o módulo principal definir uma seqüência de entrada, então emite-se uma advertência na compilação e a seqüência de entrada definida é ignorada. Se a função semântica pertencer ao caso 3 e não forem definidas seqüências de entrada ou de saída, então emite-se uma advertência

na compilação, e o código gerado utiliza a entrada e saída padrão como comportamento *default*. Um erro de execução ocorre quando as cláusulas `input` ou `output` indicam que a entrada ou a saída será definida por um identificador a ser associado na linha de comando do interpretador, e o mesmo não é realizado no momento da chamada ao interpretador.

3.3 Especificação de Domínios Sintáticos

Em *Notus* tipos são domínios de Scott [Scott, 1976]. Domínios sintáticos são assim declarados:

```
syntactic domainId +- "," ";"
```

onde `domainId +- ","` é uma lista de um ou mais nomes de domínios separados por vírgula. Identificadores de domínios são representados por sequências de letras e dígitos iniciadas por letra maiúscula. O exemplo a seguir define os domínios `Id`, `Num` e `Exp`.

```
syntactic Exp;
syntactic Id, Num;
```

Domínios sintáticos denotam domínios de *tokens* (Seção 3.5) e variáveis de gramática (Seção 3.6). Domínios de *tokens* e variáveis podem ser explicitamente declarados ou automaticamente coletados. No código da Listagem 3.1 a variável de gramática `exp` é explicitamente declarada no domínio `Exp`, e o *token* `num` é declarado sem domínio. No entanto, o compilador de *Notus* assume que `num` pertence ao domínio `Num`, convertendo o primeiro caractere de `num` para letra maiúscula. Adicionalmente, identificadores de *tokens* e variáveis podem ser "decorados" por uma sequência de dígitos.

```
syntactic Exp, Num;
token num = [0-9]+;
exp : Exp = exp "+" num | exp "-" num | num
```

Listagem 3.1: Exemplos de declaração de *tokens* e variáveis

Para variáveis e *tokens* que não têm seus domínios explicitamente declarados, o compilador de *Notus* adota a seguinte convenção:

- assume-se que variáveis ou *tokens* sem decoração pertencem ao domínio de mesmo nome da variável ou *token* com a primeira letra em maiúsculo;

- assume-se que variáveis e *tokens* com decoração pertencem ao mesmo domínio de sua versão sem decoração. Por exemplo: `num1`, `num2` e `num` pertencem ao domínio `Num`.

Essa convenção adotada por *Notus* possibilita a escrita de especificações mais compactas. A legibilidade da especificação não é prejudicada, pois a convenção adotada é simples e uniforme.

Notus possui os seguintes domínios pré-definidos:

- *Bool*: representa o conjunto `false`, `true`;
- *Int*: representa o domínio dos números inteiros de 32 *bits*;
- *Long*: representa o domínio dos números inteiros de 64 *bits*;
- *Float*: representa o domínio dos números de ponto flutuante de 32 *bits*;
- *Double*: representa o domínio dos números de ponto flutuante de 64 *bits*;
- *Char*: representa o domínio dos caracteres;
- *String*: representa o domínio de todas as sequências de caracteres entre aspas duplas;

3.4 Especificação de Domínios Semânticos

Domínios semânticos em *Notus* definem o universo de valores usados na especificação da semântica de linguagens de programação. Os domínios semânticos são declarados como a seguir:

```
visibility domainName = domainExpression";"
```

O identificador do domínio semântico, assim como o identificador de domínio sintático, começa com letra maiúscula. Domínios semânticos possuem controle de visibilidade e, portanto, podem ser explicitamente declarados como públicos ou privados; na ausência de declaração explícita de visibilidade, são visíveis somente pelos módulos de seu pacote. A Listagem 3.2 exemplifica a definição dos domínios semânticos `Value`, `Loc` e `State`.

```
public Value = Int | Bool;
public Loc = Int;
public State = Loc -> (Value | {unused});
```

Listagem 3.2: Exemplos de declaração de domínios semânticos

Elementos de um domínio semântico declarado são obtidos usando-se a mesma convenção adotada para domínios sintáticos. Assim, os identificadores `value0`, `loc2`, e `state`, são, respectivamente, elementos dos domínios `Value`, `Loc` e `State`.

As próximas seções detalham como podem ser construídas as expressões que definem os domínios semânticos.

3.4.1 Expressão de Domínio

Os domínios semânticos são definidos por meio de uma expressão composta por operações realizadas sobre domínios básicos. Os domínios básicos são domínios sintáticos, domínios pré-definidos (veja Seção 3.3) ou enumeração de domínios (veja Seção 3.4.1.1).

As operações usadas para a composição de domínios semânticos são:

- produto cartesiano: para construção de tupla;
- união disjunta: para construção de soma separada[1];
- recursão: para a construção de domínios recursivos, por exemplo, listas;
- construtor de domínios funcionais;

A precedência dos elementos e operações que compõem expressões de domínio obedece a Tabela 3.2.

maior precedência	tupla, domínios pré-definidos, identificadores, enumeração
	lista
	função
menor precedência	união disjunta

Tabela 3.2: Precedências das operações e elementos da expressão de domínio

3.4.1.1 Domínio Enumeração

Domínio enumeração define conjuntos de constantes, e cada elemento de uma enumeração é um identificador iniciado com letra minúscula. Por exemplo, a Listagem 3.3 mostra a definição dos domínios `Color` e `Language`;

```
public Color = {red, green, blue};
Language = {java, haskell, notus};
```

Listagem 3.3: Exemplos de declaração de domínios semânticos usando enumeração

[1]Do inglês: "disjoint sum".

3.4.1.2 Domínio União Disjunta

O domínio união disjunta representa soma separada de domínios e possui a seguinte forma:

$$d_1 \mid d_2 \mid \cdots \mid d_n \,,$$

onde cada d_i, $0 < i \leq n$, é um domínio. A Listagem 3.4 exemplifica a criação de domínios usando união disjunta. O domínio `Value` é a soma separada dos domínios `Int`, `Bool` e {`error`}

```
Value = Int | Bool | {error};
```

Listagem 3.4: Exemplos de declaração de domínios semânticos usando tuplas

3.4.1.3 Domínio Tupla

Domínio tupla define o produto cartesiano dos domínios que o compõem. A Listagem 3.5 é um exemplo de criação de tuplas de domínios.

```
Pair = (Int, Bool);
AtMostThree = None ()|One (Int)|Two (Int, Int)|Three (Int, Int, Int);
Tree = Leaf Int | Branch (Int, Tree, Tree);
```

Listagem 3.5: Exemplos de declaração de domínios semânticos usando tuplas

As tuplas são identificadas pelo seu construtor de tipo, e seu domínios podem ser recursivamente definidos. Se um domínio é definido por apenas uma expressão tupla, seu construtor pode ser omitido. O domínio `Pair` representa `Int` $\times$ `Bool` e o construtor da tupla é `Pair`. O domínio `AtMostThree` representa $() + $ `Int` $+ ($`Int` $\times$ `Int`$) + ($`Int` $\times$ `Int` $\times$ `Int`$)$. O domínio `Tree` define recursivamente uma árvore binária não-vazia.

3.4.1.4 Domínio Lista

Um domínio lista representa listas de elementos de um domínio. A Listagem 3.6 exemplifica a criação de lista de domínios. O domínio `Input` é uma lista de valores pertencentes ao domínio `String`.

```
Input = String*;
```

Listagem 3.6: Exemplos de declaração de domínios semânticos usando lista

3.4.1.5 Domínio Funcional

Domínio funcional é uma função de um domínio para outro e possui a seguinte forma:

$$d_1 \text{ -> } d_2 \,,$$

onde d_1 e d_2 são domínios. A Listagem 3.7 exemplifica a criação de domínios funcionais.

```
State = Loc -> (Value|{error});
Econt = Value -> State -> State;
```

Listagem 3.7: Exemplos de declaração de domínios semânticos usando funções

O operador → é associativo à direita, e, dessa maneira, o domínio `Econt` representa o domínio `Value` → (`Store` → `Store`).

3.5 Especificação Léxica

Na seção de especificação léxica de um módulo em *Notus*, definem-se *tokens* da linguagem descrita. *Notus* permite a definição de macros para melhoria de legibilidade e economia de escrita. Macros, identificadas pela palavra reservada *element*, são abreviaturas de expressões regulares e podem ser usadas para a definição dos *tokens* da linguagem. *Tokens* e macros são definidos por expressões regulares e podem possuir controle de visibilidade pública ou privada, ou seja, podem ser visíveis por todos os módulos da especificação ou somente dentro do módulo em que estão definidas. Em *Notus*, macros não podem ser recursivas, pois isso permitiria a escrita de linguagens não-regulares, que não podem ser reconhecidas por autômatos finitos.

Em *Notus*, os *tokens* podem pertencer a domínios sintáticos. Na descrição de um *token* é possível determinar como seu lexema será interpretado, especificando-se uma função a ser aplicada ao lexema quando este for reconhecido pelo analisador léxico. Essa função somente se aplica aos *tokens* com domínios sintáticos definidos, e recebe como argumento um *string* e retorna um valor do seu domínio sintático.

A Listagem 3.8 exemplifica a especificação léxica em *Notus* de uma linguagem que possui identificadores e números. Os números são definidos como uma sequência de dígitos, e identificadores são representados por sequências de letras e dígitos iniciadas por letras. Conjuntos de caracteres são definidos pelos símbolos "["e "]", e intervalo de caracteres pode ser feito separando os caracteres por "-", assim [A-Z] indica o conjunto de caracteres entre "A"e "Z". As macros `letter` e `digit` e os *tokens* `id` e `num` são visíveis para todos os módulos da especificação da linguagem, ao passo que a macro

`seqId` só pode ser usada dentro do módulo onde é definida, pois não foi declarada como pública. A função `asInteger` presente na definição do *token* `num` pertence ao conjunto de funções do núcleo de *Notus* e indica que o lexema do *token* `num` será tratado como inteiro.

```
1 public token id : Id = seqId ;
2 public token num : Num = digit+ is asInteger ;
3 element seqId = letter ( letter | digit ) * ;
4 public element letter = [A–Z] | [a–z] ;
5 public element digit = [0–9] ;
```

Listagem 3.8: Exemplo de Especificação Léxica de uma Linguagem em *Notus*

A cláusula *ignore* de *Notus* define elementos a serem ignorados durante a análise léxica, tais como espaços em branco e comentários, por meio de uma expressão regular. O exemplo da Listagem 3.9 indica que espaços em branco devem ser ignorados durante a análise léxica da linguagem especificada, bem como qualquer sequência de caracteres que inicie em "/*" e termine em "*/", ou que inicie em "//".

```
1 ignore " " | comments ;
2 element comments = "//" .* | "/*" ( . | "\n" )* "*/" ;
```

Listagem 3.9: Exemplo de uso da cláusula *ignore* de *Notus*

Notus permite a extensão de definições de *tokens* e macros por meio da união de uma nova expressão regular. A extensão de *tokens* e macros é realizada usando-se a cláusula *extend*. Para exemplificar, considere o código da Listagem 3.10 contendo a declaração dos *tokens* `num` e `id`.

```
1 token num : Num = [0–9]+ is getNumber ;
2 token id : Id = [a–z]+[0–9]* is getId ;
```

Listagem 3.10: Exemplo de definição de *tokens*: `id` e `num`

A Listagem 3.11 mostra a extensão dos *tokens* `num` e `id`, fazendo com que a linguagem definida suporte números em hexadecimal e permita que identificadores comecem com o caractere "_".

```
1 extend token num with "0x"[0–9A–Fa–f]+ is getNumberInHexa ;
2 extend token id with "_"[a–z]+[0–9]* ;
```

Listagem 3.11: Exemplo extensão de *tokens*

Na declaração de uma extensão de *tokens* não é possível redefinir seu domínio. É possível, entretanto, especificar uma nova função a ser aplicada ao lexema quando este

for reconhecido pelo analisador léxico. Dessa maneira, quando um *string* *s* for reconhecido pela expressão regular [0-9]+, a função `getNumber` é aplicada a *s*. Por outro lado, se *s* for reconhecido pela expressão regular "0x"[0-9A-Fa-f]+, a função getNumberInHexa é aplicada a *s*. Quando nenhuma nova função é especificada, como é o caso da extensão do *token* `id` na Listagem 3.11, se o string *s* for reconhecido pela expressão regular "_"[a-z]+[0-9]* a função getId será aplicada a *s*.

Extensão de macros é feita de forma semelhante à de *tokens*. A Listagem 3.12 exemplifica a extensão da macro *letter*. Inicialmente, `letter` denota apenas letras minúsculas e, após a extensão, denota também letras maiúsculas.

```
1 element letter = [a–z] ;
2 extend element letter with [A–Z] ;
```

Listagem 3.12: Exemplo de extensão de macros

A definição de *letter*, definida na Listagem 3.12, é equivalente a:

```
element letter = [a–z] | [A–Z] ; // or [a–zA–Z]
```

3.6 Especificação Sintática

O projetista deve definir os nomes dos domínios sintáticos e a gramática concreta da linguagem, possibilitando ao compilador *Notus* a geração da gramática abstrata e dos *datatypes* em *Haskell* que representam estes domínios sintáticos.

Variáveis de gramática são definidas por meio de declarações de variáveis e, de forma semelhante aos *tokens*, podem possuir visibilidade pública ou privada. Note que para a declaração de um *token* utiliza-se a palavra reservada *token*, mas a declaração de variáveis dispensa o uso de palavra reservada, evitando-se a escrita verbosa de uma gramática. Para a visibilidade pública usa-se a palavra reservada *public*, e para a visibilidade privada usa-se a palavra *private*. Variáveis sem definição de visibilidade possuem visibilidade de pacote. Na Listagem 3.13 a variável `exp` possui visibilidade pública, a variável *factor* possui visibilidade privada, e *term* possui visibilidade só dentro do pacote.

```
1 public exp : Exp ::= exp "+" term | exp "-" term | term;
2 term : Exp ::= term "*" factor | term "/" factor | factor;
3 private factor : Exp ::= id | "(" exp ")" | num ;
```

Listagem 3.13: Exemplo de regras de produção para uma linguagem aritmética

Para a descrição da gramática concreta são declaradas regras de produção. Na Listagem 3.13 é mostrado um conjunto de regras de produção para uma linguagem

aritmética simples. Os *tokens* `num` e `id` usados nesse exemplo são aqueles definidos na Listagem 3.8. Do lado esquerdo de cada produção observa-se a declaração de uma variável, e cada nome presente no lado direito deve identificar um *token* ou uma variável. Símbolos entre aspas duplas são *tokens* anônimos automaticamente incluídos na definição léxica da linguagem. Portanto, na Listagem 3.13 os símbolos "+", "-", "*", "/", "(", ")"são automaticamente adicionados ao conjunto de *tokens* da linguagem e dispensam declaração explícita para representá-los. É importante observar que *tokens* anônimos não possuem domínio.

O símbolo inicial da gramática é definido por meio da cláusula *startsymbol*. O módulo principal da especificação de uma linguagem deve obrigatoriamente possuir a declaração do símbolo inicial da gramática. O símbolo inicial deve ser uma variável de gramática declarada no módulo principal ou uma variável pública declarada em um dos módulos que ele importa. Cada módulo da descrição pode conter no máximo um símbolo inicial. O módulo principal é assim definido:

```
module Main
... // module constituents
startsymbol program;
... // other module constituents
end
```

A gramática abstrata de uma linguagem é derivada, pelo compilador de *Notus*, a partir da especificação da gramática concreta ou, alternativamente, definida pelo projetista anexando-se uma regra abstrata para cada regra concreta da gramática. Essas duas formas de construção da gramática abstrata da linguagem são detalhadas na Seção 4.2.

Assim como *Notus* permite a extensão de definições de *tokens* e macros, é possível também adicionar novas regras de produção a variáveis já definidas. A extensão de *tokens*, macros e variáveis permite a construção de definições modulares e incrementais de forma que extensões sejam realizadas por novos módulos, sem a necessidade de alterar os módulos existentes.

Por exemplo, a Listagem 3.14 mostra a definição de uma linguagem aritmética feita modularmente. No primeiro trecho são definidas apenas as operações de adição e multiplicação. Em um momento posterior, possivelmente em outro módulo, são definidas as operações de subtração por meio da extensão da variável `exp`, seguida da extensão da variável `term` com a inclusão da operação divisão. E por último acrescentou-se à variável `factor` uma produção para `exp` entre parênteses.

```
1 public exp : Exp ::= exp "+" term | term;
```

```
2 public term : Exp ::= term "*" factor | factor;
3 public factor : Exp ::= id | num ;
4
5                 ...
6
7 extend exp with exp "-" term;
8 extend term with term "/" factor;
9 extend factor with "(" exp ")";
```

Listagem 3.14: Exemplo de regras de produção para uma linguagem aritmética usando a cláusula *extend*

Não existe diferença semântica entre regras definidas no momento da declaração da variável e regras definidas por meio de cláusula *extend*. Dessa maneira, nenhuma modificação é necessária na especificação da gramática abstrata ou dos domínios sintáticos.

3.7 Especificação Semântica

A especificação semântica em *Notus* é composta pelos domínios semânticos (Seção 3.4), pelas funções semânticas, e pelas equações que descrevem a semântica dos construtos da linguagem de programação especificada. As equações semânticas são descritas por funções. Para definir uma função semântica em *Notus*, o projetista deve declarar sua assinatura (veja Seção 3.7.1) e então definir o corpo da função, escrevendo pelo menos uma equação de definição (veja Seção 3.7.2). As equações de definição para uma função determinam o resultado de sua aplicação para valores que casam com um conjunto de padrões em particular e podem ser dadas em qualquer ordem e em módulos distintos.

3.7.1 Declaração de Função

Uma declaração de função define sua assinatura por meio da palavra reservada `function`, seguida pelo nome da função com a primeira letra em minúsculo, e por uma expressão de domínio (veja Seção 3.4.1). A Listagem 3.15 mostra um exemplo de declaração de uma função de denotação de expressões `denExp`. As funções podem ser declaradas com controle de visibilidade. Uma função pode ser pública, quando declarada com a palavra-chave *public*, privada, quando declarada com a palavra-chave *private*, ou ser visível pelos módulos do mesmo pacote na ausência do qualificador de controle de visibilidade.

```
1 public function denExp : Exp -> Value;
```

Listagem 3.15: Exemplo de declaração de função.

3.7.2 Definição de Função

O corpo de uma função é descrito por pelo menos uma equação de definição de função, composta pelo nome da função seguido pelos parâmetros e por uma expressão (veja Seção 3.7.4). Os parâmetros das definições de funções são padrões (veja Seção 3.7.3) a serem casados com os argumentos de uma expressão de aplicação de função (veja Seção 3.7.5.4). A Listagem 3.16 mostra definições para as funções `fact` e `pow`. A função `fact`, quando aplicada ao valor 0, gera como resultado 1. Caso contrário o argumento de chamada é ligado ao parâmetro `n`, e a expressão `n * fact (n - 1)` é avaliada. Não se pode usar o mesmo identificador para dois parâmetros em uma mesma definição de função.

```
1 function fact : Int -> Int;
2 fact 0 = 1;
3 fact n = n * fact (n - 1);
4
5 public function pow : Int -> Int -> Int;
6 pow a 0 = 1;
7 pow a n = a * pow a (n - 1);
```

Listagem 3.16: Exemplo de definição de função.

3.7.3 Padrões

Padrões são utilizados para a realização de casamento de padrão, onde o valor de uma expressão é comparado à estrutura do padrão. Padrões são usados em:

- definições de funções, como parâmetros a serem casados com os argumentos de uma expressão de aplicação de função;
- expressões de casamento de padrão, para inspeção de tipo de uma expressão;
- variáveis das expressões *let*, *where*, *case* e abstração lambda.

Os padrões existentes em *Notus* são valor literal, identificador, agregado tupla, agregado lista, e nó de *AST*.

Padrões literais. Padrões literais casam apenas com seus respectivos valores, como é o caso do padrão 0 da primeira definição de `fact` da Listagem 3.16.

Padrões Identificadores. Os padrões identificadores podem casar com quaisquer valores respeitando as possíveis variações definidas a seguir. Esses padrões podem declarar:

- *elementos de enumeração*, que casam apenas com o elemento da enumeração a que se referem;
- *identificadores decorados representando elementos de domínios*, que casam apenas com valores do domínio que representam. O domínio é obtido ignorando-se a decoração e capitalizando-se a primeira letra do identificador;
- *identificadores não-declarados*, que, a princípio, podem representar elementos de quaisquer domínios especificados, casam apenas com valores de um domínio dependente do contexto em que se encontram;
- *identificadores anônimos*, representados pelo caractere '_', casam com qualquer valor. Esses padrões não são associados ao valor casado.

A função `sum` da Listagem 3.17 exemplifica o uso de identificadores elementos de enumeração com o padrão `error`, o uso de identificadores decorados de elementos de domínio, com os padrões `int1` e `int2`, e o uso do identificador anônimo `'_'`.

```
1 Value = Int | {error};
2
3 function sum : Value -> Value -> Value;
4 sum int1 int2 = int1 + int2;
5 sum error _ = error;
6 sum _ error = error;
7 sum _ _ = error;
```

Listagem 3.17: Exemplo de padrões identificador em definição de função.

Os identificadores `a` e `n` nas definições das funções `fact` e `pow` da Listagem 3.16 são exemplos de padrões com identificadores não-declarados, ou seja, seu tipo é inferido pelo contexto.

Padrão Agregado Tupla. O padrão tupla casa com valores de um domínio tupla é representado pelo construtor da tupla seguido pelos seus elementos entre parênteses e separados por vírgula. A Listagem 3.18 exemplifica a utilização de padrões de tupla em definições de função. O domínio `P` representa um par ordenado com coordenadas inteiras. A função `quadrant` determina qual é o quadrante do plano cartesiano em que se encontra um determinado ponto representado por um par ordenado no domínio `P`. Se as coordenadas forem ambas iguais a 0, o par está na origem do plano cartesiano. Caso contrário, a função `quadrant` testa os valores do par determinando sua posição.

```
1  P = (Int,Int);
2  function quadrant : P -> String;
3  quadrant P(0,0) = "origin";
4  quadrant P(x,y) = if (x>0 and y>0)
5                       then "quadrant 1"
6                       else if (x<0 and y<0)
7                         then "quadrant 3"
8                         else if (x>0 and y<0)
9                           then "quadrant 2"
10                          else "quadrant 4";
```

Listagem 3.18: Exemplo de padrão tupla em definição de função.

Padrão Agregado Lista. O padrão lista casa com valores lista determinados pelo casamento seqüencial, ou pelo casamento com identificadores de lista decorados, ou por casamento de cabeça/cauda (*head/tail*) de lista.

No casamento seqüencial os elementos são listados entre parênteses e separados por vírgula. Na Listagem 3.19, a função `listP` mapeia uma lista de inteiros em uma nova lista de inteiros de acordo com os elementos da lista de argumento; se o argumento for uma lista vazia então uma lista vazia é retornada; senão, se o argumento é a lista `(1,2,3)` então a lista `(1,2,5)` é retornada; senão se o argumento é uma lista com três elementos, e o primeiro é 1 e o último é 3, então é retornada a lista argumento substituindo-se apenas o último elemento para 4; e, finalmente, se o argumento é uma lista em que o primeiro elemento é 1 e o segundo e quarto são ambos 2, então a lista `(1,2,2)` é retornada. De forma análoga, a função `listP2` mapeia qualquer lista no formato `((1), (), (x,y))` em uma lista `(1,x + y)`.

```
1  function listP : Int* -> Int*;
2  listP () = ();
3  listP (1,2,3) = (1,2,5);
4  listP (1,x,3) = (1,x,4);
5  listP (1,2,_,2) = (1,2,2);
6
7  function listP2 : Int** -> Int*;
8  listP2 ((1), (), (x,y)) = (1,x + y);
```

Listagem 3.19: Exemplo de padrão de lista com casamento seqüencial em definição de função.

O casamento com identificadores de lista decorados ocorre quando um identificador é decorado pelos símbolos `*` ou `+`. Na Listagem 3.20 a definição para a função `listP3` utiliza o padrão de lista decorado `bool*` para representar uma lista de valores booleanos que, quando casada, gera como resultado a lista `(true,true)`.

```
1 function listP3 : Bool* -> Bool*;
2 listP3 bool* = (true, true);
```

Listagem 3.20: Exemplo de padrão de lista utilizando casamento com identificadores de lista decorado em uma definição de função.

No casamento de cabeça/cauda, uma lista é descrita por um padrão para o seu primeiro elemento (cabeça), e outro para os demais elementos (cauda), separados por dois pontos. A Listagem 3.21 mostra definições para a função `listP4`, que mapeia lista de inteiros em outra, utilizando padrões de cabeça/cauda de listas. A primeira definição apenas mapeia lista vazia em lista vazia. Na segunda definição casam-se argumentos de tipo lista cujo primeiro elemento é o valor 3. As demais definições de `listP4` mostram diferentes padrões de lista.

```
1 function listP4 : Int* -> Int*;
2 listP4 () = ();
3 listP4 3:s* = 4 : listP4 s*;
4 listP4 1:2:s* = 1:2: listP4 s*;
5 listP4 int:ints* = int * int : listP4 ints*;
```

Listagem 3.21: Exemplo de padrão de lista com casamento cabeça/cauda em definição de função.

Padrão Nodo de *AST*. O padrão de nodo de *AST* (Árvore de sintaxe abstrata) corresponde ao lado direito de uma regra de gramática abstrata existente. Esse padrão é definido entre colchetes, onde podem aparecer apenas padrões de identificadores, *strings* entre aspas, e padrões identificador de lista. Por exemplo, considere especificação da Listagem 3.22, cuja gramática abstrata é:

```
Exp → Exp "+" Exp
    | Exp "*" Exp
    | Int
```

A primeira definição determina o resultado da função de denotação de expressões `dExp` para um argumento inteiro; a segunda determina o resultado de `dExp` para uma expressão de adição; e a última determina o resultado de `dExp` para uma expressão de multiplicação.

```
1
2  token num : Exp = [0-9]+ is asInteger;
3
4  exp ::= exp "+" term | term : term;
5  term : Exp ::= term "*" factor | factor : factor ;
6  factor : Exp ::= num | "(" exp ")" : exp;
7
8  function dExp : Exp -> Int;
9  dExp int = int;
10 dExp [exp1 "+" exp2] = dExp exp1 + dExp exp2;
11 dExp [exp1 "*" exp2] = dExp exp1 * dExp exp2;
```

Listagem 3.22: Exemplo de padrão de nodo de *AST* em definição de função.

3.7.4 Expressões

As expressões existentes em *Notus* são brevemente descritas nas próximas seções, e detalhadas no manual da linguagem em [Tirelo e Bigonha, 2006].

3.7.5 Valores Literais

Valores literais em *Notus* são definidos para cada domínio pré-definido da Seção 3.3.

3.7.5.1 Agregados

Agregado tupla é formado por um construtor e uma seqüência de expressões, separadas por vírgula, e entre parênteses. Mostram-se a seguir exemplos de agregados tupla, onde `A, B, C, D` e `E` são construtores.

```
A() B(true) C(1,true,3 + 7)
D(false,3.141592,E(a,if b then 10 else 9))
```

Agregado lista é uma seqüência de expressões entre parênteses e separadas por vírgula. A seguir, são mostrados exemplos de agregados lista.

```
() (true) (1,4,3 + 7)
((4,3),(a,if b then 10 else 9))
```

3.7.5.2 Identificadores

Identificadores em *Notus* são seqüências de letras, dígitos e caracteres '_', iniciados com letra minúscula. Identificadores podem ser qualificados com o nome do módulo em que

estão definidos, e decorados com números ou como lista, por meio do sufixo `*`. O domínio de um identificador é obtido automaticamente, capitalizando-se sua primeira letra e ignorando-se sua decoração.

3.7.5.3 Operações

Notus possui operadores básicos para realização de operações aritméticas *bit-a-bit*, relacionais, booleanas e manipulações de listas. No manual de *Notus* [Tirelo e Bigonha, 2006], encontra-se uma tabela onde se relaciona cada operador pré-definido seguido pelo seu uso, sua precedência e associatividade.

3.7.5.4 Aplicação de Função

Aplicações de função possuem a forma:

$$e_0\ e_1\ e_2\ \cdots\ e_n$$

onde cada e_i é uma expressão. Se e_i possui domínio d_i, para cada $i \geq 1$ e e_0 possui domínio $d_1 \rightarrow d_2 \rightarrow \cdots \rightarrow d_n \rightarrow d$, então a aplicação de função $e_0\ e_1\ e_2\ \cdots\ e_n$ possui domínio d.

Os argumentos de uma aplicação de uma função `f` definem qual das definições existentes para `f` será usada. Seja então uma função `f` de assinatura $d_1 \rightarrow d_2 \rightarrow \cdots \rightarrow d_n \rightarrow d$, com corpo composto por m definições de funções, cada uma delas composta pelo conjunto de padrões $< p_{k1}, p_{k2}, \cdots, p_{knj} >$, com $1 \leq k \leq n_j$. Note que, cada definição j de uma mesma função pode variar o número de parâmetros explícitos. A avaliação dos argumentos da aplicação de função $f_0\ e_1\ e_2\ \cdots\ e_n$ deve casar com alguma das m definições de `f` de acordo com o seu conjunto de padrões. Caso isso não ocorra, um erro de execução é gerado.

Se o último argumento da aplicação de função for complexo e demandar o uso de parênteses, esses podem ser substituídos pela cláusula *evaluate expresssion* de *Notus*, que possui a seguinte forma:

$$\textbf{evaluate}\ \{\ function; \mathrm{arg}_1; \mathrm{arg}_2; \cdots; \mathrm{arg}_{n-1}; \mathrm{arg}_n\ \}$$

Essa expressão é equivalente a: $function(\mathrm{arg}_1(\mathrm{arg}_2(\cdots(\mathrm{arg}_{n-1}(\mathrm{arg}_n)))))$. Por exemplo, a expressão `evaluate { fun a b; c d; e f }` é equivalente a `fun a b (c d (e f))`.

3.7.5.5 Casamento de Padrão

A expressão de casamento de padrão executa apenas uma inspeção de tipo e, portanto, não associa o valor da expressão ao padrão testado. Logo, o padrão casado não pode ser usado fora da expressão de casamento de padrão. Essa expressão é da forma:

```
e is p
```

onde `e` é uma expressão, cujo domínio será casado com o padrão `p`. O resultado dessa expressão é um valor booleano.

A Listagem 3.23 exemplifica essa expressão. A função `pExp` utiliza a expressão de casamento de padrão para testar se o argumento `value` pertence a um dos domínios `Int, Bool` ou `error`.

```
1 Value = Int | Bool | {error};
2
3 function pExp : Value -> String;
4 pExp value = if value is int then "int value"
5              else if value is bool then "bool value"
6              else "error";
```

Listagem 3.23: Exemplo de expressão de casamento de padrão.

3.7.5.6 Abstração Lambda

Abstrações lambda têm a forma

$$\backslash p_1\ p_2 \cdots\ p_n \rightarrow e$$

onde cada p_i é um padrão e e é uma expressão. Na listagem 3.24, a definição para a função `add` retorna uma função, utilizando abstração lambda, que retorna a soma de dois inteiros.

```
1 function add : Int -> Int -> Int;
2 add = \int1 int2 -> int1 + int2;
```

Listagem 3.24: Exemplo de abstração lambda.

3.7.5.7 Expressões *Let* e *Where*

Expressões *let* e *where* são utilizadas para introduzir definições locais auxiliares. A Listagem 3.25 apresenta funções que calculam o *n-ésimo* número da série de Fibonacci utilizando expressões *where* e *let*.

```
1 function wherefib : Int -> Int;
2 wherefib 0 = 0;
3 wherefib 1 = 1;
4 wherefib n = x + y where {x = wherefib (n-1); y = wherefib (n-2)};
5
6 function letfib : Int -> Int;
7 letfib 0 = 0;
8 letfib 1 = 1;
9 letfib n = let {x = letfib (n-1); y = letfib (n-2)} in  x + y;
```

Listagem 3.25: Exemplos de expressões *let* e *where*.

3.7.5.8 Função de Atualização

A expressão de atualização de função em *Notus* é usada para atualizar determinados pontos de uma função. Os pontos da função e seus novos valores são definidos na expressão de atualização. Essa função tem a forma

$$f[a \leftarrow v]$$

onde f é uma função que aplicada ao argumento a retorna o novo valor definido v, e, aplicada a qualquer outro valor produz o mesmo resultado de f.

Por exemplo, na Listagem 3.26, a função `fact` calcula o fatorial de um número, e a função g é definida por:

$$g\ x\ = \begin{cases} 5, \text{ se } x = 1 \\ \text{fact}\ \ x, \text{ caso contrário} \end{cases}$$

```
1 function fact : Int -> Int;
2 fact 0 = 1;
3 fact n = n * fact (n - 1);
4
5 function g : Int -> Int;
6 g = fact[1<-5];
```

Listagem 3.26: Exemplo de expressão de atualização de função.

3.7.5.9 Expressão *If*

A expressão *if* possui a forma:

$$\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3$$

onde e_1 é uma expressão booleana e e_2 e e_3 são expressões pertencentes a domínios compatíveis (veja Seção 5.5). O valor dessa expressão é e_2 se a avaliação de e_1 resultar em *true*, ou e_3 caso contrário.

3.7.5.10 Expressão *Case*

A expressão *case* possui a forma:

$$\textbf{case}\ e_0\ \textbf{of}\ \{p_1 \rightarrow e_1; \cdots; p_n \rightarrow e_n\}$$

onde cada e_i é uma expressão, e cada p_i é um padrão. O valor da expressão *case* é e_i se o resultado da avaliação de e_0 casar com o padrão p_i. Na Listagem 3.27, a função `caseExp` testa os padrões possíveis para um dado elemento do domínio `Value`. Note que o caso *default* é obtido por meio do padrão anônimo '_'.

```
1 Value = Int | Bool | {error};
2
3 function caseExp : Value -> String;
4 caseExp value = case value of {
5                     int  -> "int value";
6                     bool -> "bool value";
7                     _    -> "error"};
```

Listagem 3.27: Exemplo de expressão *case*.

3.8 Definição de Contextos

Notus permite que informações de contexto sejam transmitidas de forma transparente entre as construções de uma especificação. Por exemplo, para uma especificação de uma linguagem em que a definição semântica para algumas construções utilizam *stores* e *environments*, a equação para valores literais não precisa referenciar esses contextos, tornando a propagação de informação de contexto inconveniente. Existem também

construções, tais como comandos condicionais, cuja semântica não usa diretamente esses contextos, mas precisa propagá-los para a avaliação de suas partes.

Certos recursos usuais de linguagens de programação, tais como verificação de tipos dinâmicos, tratamento de erros e seqüenciadores influenciam o comportamento de outros, e esta influência deve ser explicitada nas equações semânticas que os definem. Para exemplificar a dependência semântica entre construções de uma linguagem, considere o código da Listagem 3.28. O retorno R_1 deve executar o bloco C_3 antes de efetuar o retorno de `f`. Por outro lado, o retorno R_2 é executado imediatamente. Assim, a definição semântica do comando *try-catch-finally* é influenciada pela semântica de métodos e retorno de método. Adicionalmente, o comando **return** é dependente do contexto em que se encontra, assumindo semânticas diferentes dependendo do seu local de ocorrência na árvore de sintaxe abstrata do programa.

```
1  int f () {
2    try {
3      C1 // calculation of result
4      if (...) return result ; // R1
5      C1'
6    } catch (Exception e) {
7        C2 // exception handling
8    } finally {
9        C3 // some operation to finalize execution
10   }
11   return default_value ; // R2
12 }
```

Listagem 3.28: Comando *try-catch-finally*.

As próximas seções apresentam as construções de *Notus* que manipulam informações de contexto conforme as características das equações de um construto.

3.8.1 Domínios de Contexto

Domínios de contexto são definidos por declarações de domínios semânticos e consistem em tuplas constituídas por informações de contexto e rotuladas com os nomes das informações que se deseja propagar. Uma definição de contexto possui a forma:

$$\mathsf{T} = \textbf{context}(\mathsf{label}_1 : \mathsf{domain}_1,\ \mathsf{label}_2 : \mathsf{domain}_2,\ \cdots,\ \mathsf{label}_n : \mathsf{domain}_n)$$

onde cada $\texttt{label}_i$ é o rótulo de uma informação de contexto, e $\texttt{domain}_i$, o domínio do valor que compõe a informação. As informações podem ser classificadas de acordo com a forma como são propagadas. Uma informação efêmera, definida com a palavra-chave

`ephemeral` antes de seu rótulo, tem sua validade prescrita a cada atualização. Uma informação persistente, definida com a palavra-chave `persistent` antes de seu rótulo, não é invalidada mediante uma atualização. Caso nenhuma marca esteja presente na definição da informação, ela é considerada persistente.

A Listagem 3.29 declara um contexto composto por *stores*, cujo domínio é `S`, e `env`, cujo domínio é `R`. Neste exemplo, a informação `store` é efêmera, e *environment* é persistente.

```
T = context(ephemeral store : S, persistent env : R)
```

Listagem 3.29: Exemplos de declaração de domínios de contexto

3.8.2 Expressões e Padrões de Contexto

Contextos podem ser manipulados por meio de expressões de construção, atualização, oclusão e seleção.

3.8.2.1 Construtor de Contexto

A expressão de construtor de contexto define um valor em um domínio de contexto fornecendo valores iniciais, e possui a forma:

$$\texttt{T}\{\texttt{label}_1 \leftarrow \texttt{exp}_1,\ \texttt{label}_2 \leftarrow \texttt{exp}_2,\ \cdots,\ \texttt{label}_n \leftarrow \texttt{exp}_n\},$$

onde `T` é um domínio de contexto, cada $\texttt{label}_i$ é um rótulo de contexto e $\texttt{exp}_i$ é uma expressão do domínio de $\texttt{label}_i$.

A Listagem 3.30 exemplifica uma expressão de construtor de contexto que define valores iniciais para a declaração da Listagem 3.29.

```
T{store ← (\loc -> unused), env ← (\id->unbound)}
```

Listagem 3.30: Exemplo de expressão de construção de contexto

3.8.2.2 Seleção de Informação de Contexto

A expressão de seleção de contexto retorna o valor associado a uma informação a partir de seu rótulo. Essa expressão possui a forma:

```
label t
```

onde `t` pertence a um domínio de contexto e `label` é um rótulo do contexto definido pelo domínio de `t`.

Seja `t` um elemento do domínio `T` definido na Listagem 3.29, a expressão de seleção de domínio da Listagem 3.31 tem como resultado o valor do `store` associado a `t`.

```
store t
```

Listagem 3.31: Exemplo de expressão de seleção de contexto

3.8.2.3 Atualização de Contexto

A expressão de atualização de contexto define novos valores para informações de um contexto existente, criando um novo contexto. Essa expressão possui a forma:

$$\texttt{t[label}_1 \leftarrow \texttt{exp}_1\texttt{, label}_2 \leftarrow \texttt{exp}_2\texttt{, } \cdots \texttt{, label}_n \leftarrow \texttt{exp}_n\texttt{]}$$

onde `t` é um valor no domínio de contexto `T`, cada $\texttt{label}_i$ é um rótulo do conjunto de labels de `T`, e cada $\texttt{exp}_i$ é uma expressão pertencente ao domínio rotulado por $\texttt{label}_i$.

Seja `t` um elemento do domínio `T` definido na Listagem 3.29, e `s` um elemento do domínio `S`. A expressão da Listagem 3.32 define um novo contexto, onde um novo valor `s` é associado a `store`, e o `environment` original de `t` é mantido.

```
t[store <- s]
```

Listagem 3.32: Exemplo de expressão de atualização de contexto

3.8.2.4 Oclusão de Contexto

A expressão de oclusão de contexto é uma operação sobre dois contextos, $\texttt{t}_1$ e $\texttt{t}_2$, cujo objetivo é atualizar todas as informações efêmeras de $\texttt{t}_1$ a partir dos valores associados em $\texttt{t}_2$. Essa expressão possui a forma:

$$\texttt{t}_1 \texttt{ ** } \texttt{t}_2$$

onde $\texttt{t}_1$ e $\texttt{t}_2$ pertencem a um domínio de contexto `T`. O resultado dessa operação é um novo contexto `t` tal que para cada rótulo `label` de `T`, tem-se que `label t` é igual a:

- indefinido, se `label` $\texttt{t}_1$ e `label` $\texttt{t}_2$ forem indefinidos;
- `label` $\texttt{t}_1$, se `label` $\texttt{t}_2$ for indefinido ou `label` $\texttt{t}_2$ corresponder a uma informação de contexto efêmera;
- `label` $\texttt{t}_2$, caso contrário.

Por exemplo, sejam `t1 = T(store` ← s1, env ← r1) e `t2 = T(store` ← s2, env ← r2) contextos do domínio `T` definido na Listagem 3.29. O valor da expressão `t1 ** t2` é `t = T(store` ← s1, env ← r2).

3.8.2.5 Padrão de Contexto

O padrão de contexto é usado em expressões de casamento de padrão, e possui a forma:

$$\texttt{patt}\{\texttt{patt}_1 \leftarrow \texttt{label}_1,\ \texttt{patt}_2 \leftarrow \texttt{label}_2,\ \cdots,\ \texttt{patt}_n \leftarrow \texttt{label}_n\}$$

onde `patt` é um padrão de domínio ou um padrão de contexto, cada $\texttt{label}_i$ representa um rótulo, e cada $\texttt{patt}_i$ é um padrão válido para o domínio de $\texttt{label}_i$.

3.8.3 Expansão de Contexto

Em *Notus* contextos podem ser expandidos adicionando-se novas informações a um domínio de contexto existente. Assim como ocorre com as extensões de variáveis de gramática e *tokens*, a extensão de contexto é realizada por meio da cláusula `extend`, da seguinte forma:

$$\textbf{extend}\ \mathsf{T}\ \textbf{with}\ \texttt{label}_1\texttt{:domain}_1,\ \texttt{label}_2\texttt{:domain}_2,\cdots,\ \texttt{label}_n\texttt{:domain}_n$$

onde cada $\texttt{domain}_i$ é o domínio da nova informação rotulada por $\texttt{label}_i$. Opcionalmente, cada `label` pode ser marcado com as palavras-chave `persistent` ou `ephemeral`.

A Listagem 3.33 estende o contexto da Listagem 3.29 adicionando informações de entrada e saída, ambas efêmeras e representadas por listas de elementos do domínio `V`.

```
extend T with ephemeral input : V*, ephemeral output : V*
```

Listagem 3.33: Exemplo de expansão de contexto

3.8.4 Contexto Estrutural

O contexto estrutural de uma construção é formado pelas informações pertencentes aos seus nodos ancestrais na árvore de sintaxe abstrata do programa. Cada nó ancestral é associado às informações disponíveis durante a avaliação desse nodo.

Domínios de contexto armazenam informações de contexto estrutural rotuladas por *structure*. Essas informações de contexto são registros de aplicações de funções, onde cada registro armazena o nome da função e os argumentos da aplicação. Essas informações, armazenadas na pilha de registro de aplicação de função, podem ser acessadas pela expressão *match* de Notus. Essa expressão possui a forma:

match p **with** e_1 **in** e_2

onde `p` é um padrão de contexto estrutural, e e_1 e e_2 são expressões. O valor da expressão `match` é o valor de e_2 se o valor de e_1 for um valor de contexto `t` e o padrão `p` casar com pelo menos um registro de rótulo `structure` em `t` na pilha de contextos.

3.9 Transformações de Módulos

Um módulo em *Notus* pode ser transformado por meio de uma função de transformação de módulo, que cria uma *AST* a partir da *AST* original do módulo, modificando algumas de suas definições.

A transformação de módulo permite que uma especificação seja realizada de forma incremental. Por exemplo, considere uma linguagem definida usando semântica denotacional direta composta pelos módulos, $M_1, M_2, \cdots, M_k$, e que o módulo M_{k+1} define seqüenciadores, cuja definição semântica requer o uso de continuações. Para evitar que os módulos existentes tenham que ser reescritos, é suficiente aplicar uma transformação de módulos por meio da definição de uma função transformadora no módulo M_{k+1}. A função transformadora pode alterar a assinatura das funções e equações semânticas de forma que, por exemplo, essas passam a incorporar continuações.

Transformadores são funções *Notus* de domínio `Module` $\rightarrow$ `Module`, onde `Module` é um domínio pré-definido cujos valores são módulos de uma especificação *Notus*, ou subtipos de `Module`, que representam os componentes de um módulo *Notus*. Uma função transformadora cria um novo módulo, a partir de um módulo inicial, processando individualmente seus componentes.

As transformações de módulos são especificadas na lista de importação de um módulo. Por exemplo, se um módulo M_1 importa o módulo M_2, aplicando a função de transformação `t`, então a lista de importação de M_1 contém `f` M_1. No exemplo da Figura 3.2, o módulo `Ma` define a função `f`, e o módulo `Mb` define a função `g`. O módulo `Mc` define uma função de transformação de nome `t`, que é aplicada aos elementos do módulo `Ma`. O módulo `Md` aplica essa mesma transformação, importada do módulo `Mc`, aos elementos do módulo `Mb`.

A seguir é apresentado um exemplo de definição de função de transformação e os seus efeitos. Inicialmente é apresentado na Listagem 3.34 a implementação em *Notus* da função de transformação `delta`, em seguida é apresentado na Listagem 3.35 um módulo de definição para uma linguagem simples, denominada *Yocto*. Na Listagem 3.36 é apresentada definição da linguagem *Zepto* que estende a linguagem *Yocto* acrescentando variáveis e comando de atribuição. Finalmente, na Listagem 3.37 são apresentados os efeitos da aplicação da função de transformação `delta` na linguagem *Yocto*.

```
module Ma
 function f : A -> B;
  f a = ...
  //...
end

module Mb
  function g : X -> Y;
  g x = ...
  //...
end
```

```
module Mc
  import t Ma, ...;
  function t : Module -> Module;
  //... usage of (f a) ...
end

module Md
  import Mc, t Mb, ...;
  //... usage of (g x) ...
end
```

Figura 3.2: Exemplo de módulos de uma especificação.

Funções de transformação operam sobre a árvore de sintaxe abstrata de um módulo e produzem uma nova árvore de sintaxe abstrata correspondente ao módulo transformado. A transformação δ define a inclusão de contexto na avaliação de expressões sem efeitos colaterais. As regras de reescrita de algumas construções fundamentais são as seguintes:

$$\delta[\![d_1 \to d_2]\!] = [\![d_1 \to \delta[\![d2]\!]]\!]$$
$$\delta[\![d]\!] = [\![t \to d]\!]$$
$$\delta[\![id]\!] = [\![\lambda t[r \leftarrow env].r\ id]\!]$$
$$\delta[\![e_1\ e_2]\!] = [\![\lambda t.(\delta[\![e_1]\!]t)(\delta[\![e_2]\!]t)]\!]$$
$$\delta[\![e]\!] = [\![\lambda t.e]\!]$$

A implementação da transformação δ é realizada pelo módulo `Delta` da Listagem 3.34. As funções auxiliares `newCtxPatt` e `newCtxInfo` retornam, respectivamente, um novo padrão de contexto e uma nova informação de contexto. A função `freshid` retorna um identificador com um nome decorado para o domínio fornecido. As funções `contextId` e `envId` produzem, respectivamente, identificadores de contexto e de *environment*. A função `envLbl` produz o *label* `env`. A função `deltaT`, utilizada na transformação de aplicação de função, cria, a partir de uma expressão `exp` e um identificador `id`, uma aplicação de função em que a função é definida pela transformação delta sobre o `exp`, e cujo argumento é o identificador de contexto `id`.

A Listagem 3.35 define uma linguagem simples, *Yocto*, que possui: valores inteiros e expressões `succ` e `pred`. Essas expressões têm como resultados o sucessor e o antecessor de um número, respectivamente.

A linguagem *Zepto*, definida na Listagem 3.36, é uma extensão de *Yocto* contendo variáveis e comando de atribuição. Devido à presença de variáveis, o contexto de avaliação de comandos e expressões considera que o contexto de avaliação é formado

```
module Delta
import Notus // Imports all Notus constructs

// Transformation of function definitions
function delta : FunDef -> FunDef
delta ["function" id dom exp] = ["function" id dom1 exp1]
  where dom1 = delta dom; exp1 = delta exp

// Transformation of domains
function delta : Dom -> Dom
// Transformation of functional domain
delta [dom1 "->" dom2] = [dom1 "->" dom3]
  where dom3 = delta dom2
// Transformation of others domains
delta dom = [dom1 "->" dom] where dom1 = contextId

function delta : Exp -> Exp

// Transformation of identifier expression
delta idExp = ["lambda" patt exp]
  where patt = newCtxPatt id1 (newCtxInfo id2 envLbl)
                                    //yields t{r <- env}
        id1 = freshid contextId (idExp) //id1 = t
        id2 = freshid envId (idExp)     //id2 = r
        exp = newFunApp id2 idExp //yields r idExp

// Transformation of function application
delta ["apply" exp1 exp2] = ["lambda" id exp3]
  where exp3 = newFunApp (deltaT exp1 id) (deltaT exp2 id)
                          //yields (delta exp1 t) (delta exp2 t)
        id = freshid contextId (exp1,exp2)
                          //yields t

// primitive expressions
delta exp = ["lambda" id exp1]
  where id = freshid contextId (exp)
```

Listagem 3.34: Modulo que implementa a função de transformação `delta`

```
module Yocto.Basics
// Ignores white spaces
ignore [ \n\r\t]

// Numbers: signed sequence of digits
public n ::= ("+"|"-")?[0-9]+ is asInteger

// Expressions: number, and successor and predecessor functions
public e ::= "succ" e | "pred" e | n

// Denotation of expressions
public function de : E -> Int
de ["succ" e] = de e + 1
de ["pred" e] = de e - 1
de [int] = int
end
```

Listagem 3.35: Módulo de especificação *Notus* para a linguagem *Yocto*.

por um ambiente que associa cada identificador a um valor inteiro. Desta forma, o domínio `T` de contextos é definido por:

```
T = context(ephemeral env : R)
```

onde `R = I →(V | {unbound})` é o domínio de funções de *environment*. Para se reutilizar o módulo `Yocto.Basics` na escrita da definição do módulo `Zepto.Basics`, deve-se aplicar, durante a importação, o transformador `delta`, definido na Listagem 3.34.

A função de transformação `delta` definida na Listagem 3.34 aplicada à função `de` do módulo `Basics`, definido na Listagem 3.35, produz o efeito ilustrado na Listagem 3.37. Assim, na definição de *Zepto*, as definições da função `de` para as expressões de `Yocto` foram transformadas para considerarem a existência de um contexto contendo um ambiente de associação de identificadores a valores.

3.10 Conclusões

Este capítulo apresentou a linguagem puramente funcional de domínio específico *Notus* para especificação de linguagens de programação. *Notus* permite a organização modular por meio de suporte a criação de módulos, pacotes e controle de visibilidade. *Notus* provê recursos para: definições léxica, e da sintaxe concreta e abstrata; definições de domínios sintáticos e semânticos; funções semânticas e definições de equações. As expressões em *Notus* são inspiradas em *Haskell*.

```
module Zepto.Basics
import Delta, // Imports the transformation
delta Yocto.Basics; // Apply delta transformer to Yocto.Basics
                    // constructions

// Define identifiers to be a sequence of at least one letter
public token id = [a-z]+

// Context domain: consider environment to be persistent
public T = context(env : R)

// Environment domain: maps identifiers to values or unbound
public R = Id -> (V|{unbound})

// Commands: only assignment is defined
public c ::= id ":=" e ";" : id ":=" e

// Denotation of commands as context transformer
public function dc : C -> T -> T
dc [id ":=" e] t[r <- env] =
  t[env <- r1] where r1 = r[id <- de e t]

// Extension of expressions to include identifiers
extend e with id ;

// Denotation of identifier: checks value in environment
de [id] t[r <- env] = r id
end
```

Listagem 3.36: Módulo de especificação *Notus* para a linguagem *Zepto*

```
function de : E -> T -> V
de ["succ" e] = \t -> de e t + 1
de ["pred" e] = \t -> de e t - 1
de [int] = \t -> int
```

Listagem 3.37: Resultado da aplicação da função de transformação `delta` sobre a função `de`

Linguagens de programação podem ser definidas em *Notus* de forma incremental, definido-se inicialmente uma linguagem base, que é sucessivamente estendida pela inclusão de novos módulos. Cada módulo pode definir a especificação léxica, sintática e semântica de construtos relacionados da linguagem.

A extensibilidade das especificações léxica e sintática em *Notus* ocorre via cláusulas `extend`, que provêem a extensão de *tokens*, macros e variáveis. Esse recurso de *Notus* permite a construção de definições modulares e incrementais de forma que extensões

sejam realizadas por novos módulos, sem a necessidade de alterar módulos existentes.

A extensibilidade das especificações semânticas em *Notus* é alcançada utilizando-se as construções para propagação de contexto e transformação de módulos. Essas construções permitem a realização de alterações em equações existentes de forma modular. Por exemplo, a transformação de uma especificação escrita em semântica denotacional direta pode ser transformada em uma especificação com semântica de continuações, por meio da inclusão de um único módulo, que define, via um transformador de módulos, como as equações existentes passam a ser tratadas.

Notus se destaca em relação às linguagens de semântica denotacional tradicional ao possibilitar a divisão em módulos com escrita incremental da especificação de uma linguagem por meio da construção `extends` e transformação de módulos. No entanto, a união dos componentes léxicos, sintáticos e semânticos de uma especificação modular para a geração de analisadores léxico, sintático e semântico únicos exige a ordenação desses componentes para que se possa resolver ambigüidades. Os próximos capítulos descrevem o processo de compilação e os problemas decorrentes da tradução de descrições léxica, sintática e semântica modulares durante a geração de código, bem como as soluções implementadas no compilador *Notus* para esses problemas.

Capítulo 4

Geração de Reconhecedores Sintáticos

O compilador da linguagem *Notus* transforma módulos que compõem uma descrição formal em um interpretador na linguagem *Haskell* para a linguagem especificada.

Para a geração dos analisadores léxicos e sintáticos para a linguagem especificada foram usados, respectivamente, os geradores de analisadores léxico e sintático *Alex* [Marlow, 2005a] e *Happy* [Marlow, 2005b], que também têm *Haskell* como linguagem alvo. Esses geradores funcionam de forma semelhante aos tradicionais *Lex* e *Yacc*. *Alex* gera, a partir da descrição dos *tokens* de uma linguagem, um módulo em *Haskell* com código para análise léxica. *Happy* gera, a partir de uma especificação da gramática da linguagem feita na notação *BNF*, um módulo em *Haskell* com código para análise sintática.

A Figura 4.1 mostra uma visão geral do funcionamento do compilador *Notus*.

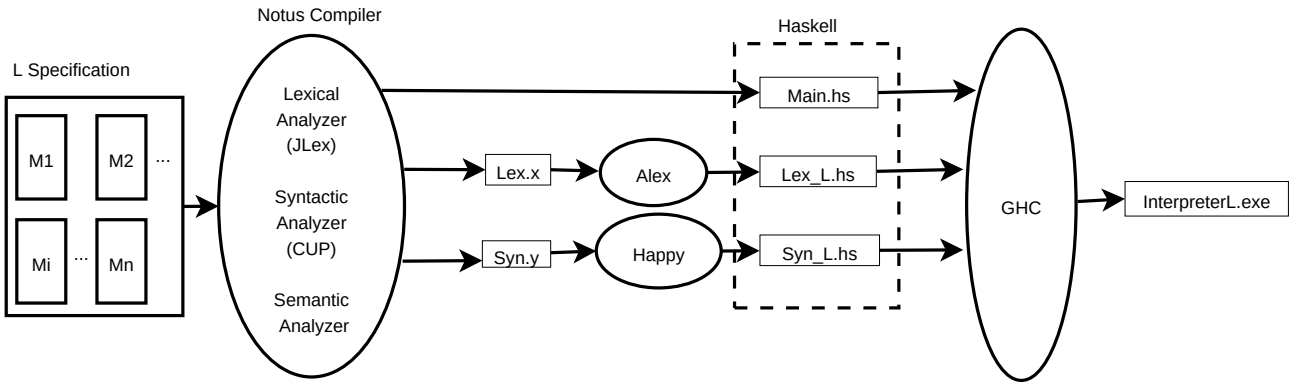

Figura 4.1: Funcionamento geral do compilador de *Notus*

O compilador de *Notus* inicialmente lê os módulos de definição da linguagem *L*, faz

as análises léxica, sintática e semântica da descrição, e gera código para a linguagem alvo *Haskell*. Os analisadores léxico e sintático desse compilador são gerados pelos programas *JLex* e *CUP*. Após a verificação léxica e sintática, a especificação de L é manipulada para a geração dos arquivos *Lex.x* e *Syn.y*, os mesmos são utilizados como entradas para os programas *Alex* e *Happy*, que geram, respectivamente, os módulos *Haskell Lex_L.hs* e *Syn_L.hs* correspondentes ao analisadores léxico e sintático de L. Além dos arquivos *Lex.x* e *Syn.y*, o compilador *Notus* gera o arquivo na linguagem alvo *Haskell Main.hs*, que é responsável por coordenar o interpretador da linguagem especificada, além de conter as funções semânticas de L. O compilador *Glasgow Haskell Compiler* (GHC) [Marlow, 2005c] faz a união e compilação desses arquivos gerando um interpretador para a linguagem L. O intepretador gerado lê programas escritos em L, faz análises léxica e sintática desses programas, interpreta-os de acordo com a semântica definida em *Notus*, e produz sua saída.

A Figura 4.2 mostra a composição dos elementos de cada módulo[1] da descrição de uma linguagem L para a construção do seu interpretador. Cada módulo da especificação de L em *Notus* descreve as especificações léxica, sintática e semântica de um conjunto de construções relacionadas. O esforço da compilação de uma especificação modular é que a união dos componentes dos módulos deve preservar as características da linguagem especificada.

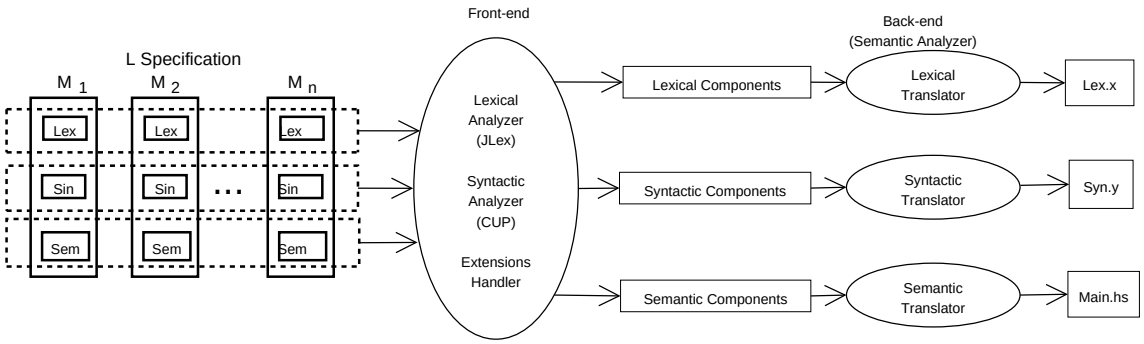

Figura 4.2: Compilação dos elementos dos módulos de uma definição em *Notus*

O *front-end* do compilador *Notus* é responsável por:

- reconhecer a especificação da linguagem L por meio das análises léxica e sintática;
- separar os componentes léxicos, dos sintáticos e dos semânticos presentes nos vários módulos da definição;
- agrupar os componentes definidos incrementalmente por meio da construção `extend`, associando-os às suas respectivas declarações.

[1]Os módulos são identificados na figura por M_1, M_2, ..., M_n

O *back-end* do compilador, dividido em três tradutores, gera os arquivos *Lex.x* e *Syn.y* no formato de entrada dos geradores de analisadores léxico e sintático *JLex* e *Happy*, e o módulo *Main.hs* contendo as equações semânticas para a interpretação de programas escritos na linguagem *L*.

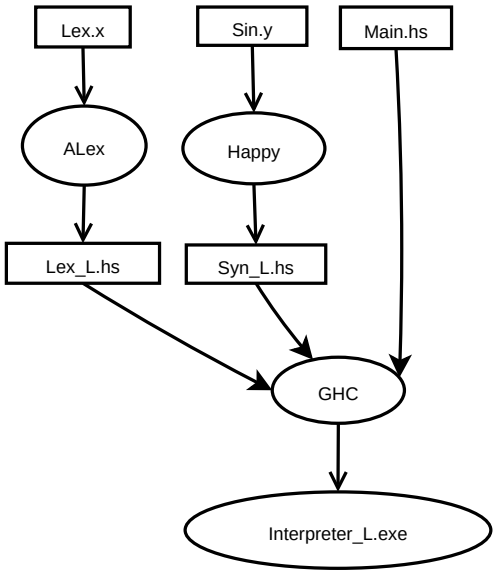

Figura 4.3: União dos analisadores léxico, sintático e semântico de *L*

O código executável, correspondente ao interpretador da linguagem *L*, é gerado por meio da compilação e ligação dos arquivos com programas *Haskell* gerados, realizada pelo compilador *GHC*. A Figura 4.3 esquematiza o processo de união dos analisadores da linguagem especificada para a obtenção de um interpretador para programas dessa linguagem.

A execução do interpretador, gerenciada pelo módulo *Main.hs* (Figura 4.3) é ilustrada na Figura 4.4. Um programa escrito em *L*, *prog.L*, é reconhecido pelo interpretador via análises léxica e sintática, e posteriormente sua estrutura é representada pela árvore de sintaxe abstrata[2] que juntamente com as entradas do programa, *input*, são interpretadas para a geração da saída, *output*.

As seções seguintes relatam a geração de reconhecedores sintáticos para linguagens especificadas detalhando-se o funcionamento dos componentes do compilador responsáveis pela geração dos arquivos de entrada para o *Alex* e *Happy* ilustrados na Figura 4.1. O capítulo 5 apresenta o processo de tradução das equações semânticas e domínios para funções em *Haskell*.

[2]AST, do inglês *abstract syntax tree*

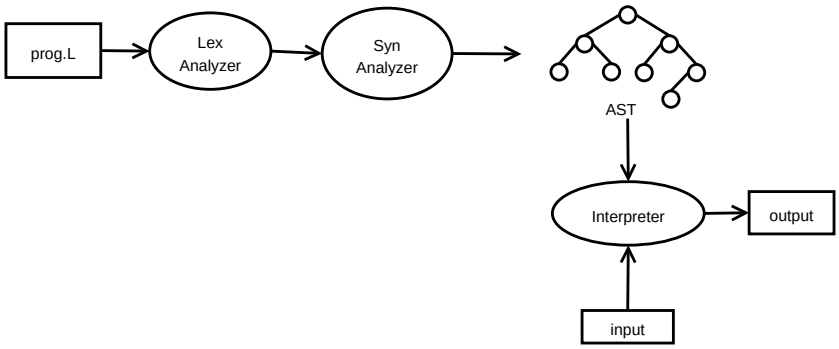

Figura 4.4: Execução do Interpretador

4.1 Compilação da Especificação Léxica

A especificação léxica de uma linguagem definida em *Notus* pode ser particionada em vários módulos. A Figura 4.5 ilustra, de forma simplificada, o processo de união das definições léxicas dos módulos em um único arquivo *Lex.x*, entrada do processador *Alex*[Marlow, 2005a], para a obtenção de um único analisador léxico da linguagem especificada. A geração de código do arquivo de entrada para o *Alex* é descrita na Seção 4.1.4.

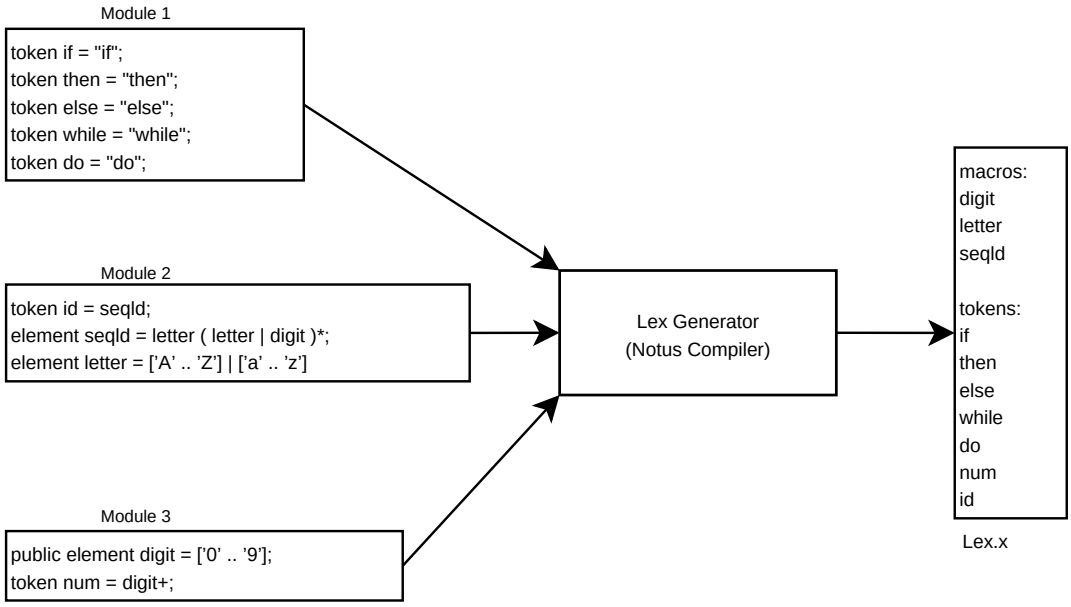

Figura 4.5: Composição da descrição léxica em um arquivo de entrada para o programa *Alex*

As etapas de tradução, ilustradas na Figura 4.6[3], realizadas pelo *back-end* do compilador *Notus* para os componentes da especificação léxica são denominadas: **Macro**

[3]Cada análise e algoritmo realizado pelo compilador de *Notus* é representado por uma caixa oval, e os dados de entrada e gerados pelos compilador são representados por retângulos.

analyzer, **Macro expansion**, **Regular expression analyzer**, **Analyze quoted**, **Analyze infinity**, **Analyze other**, **Topological sort**, e **Code generator**.

O processo de tradução opera inicialmente sobre as macros definidas nos vários módulos na etapa **Macro analyzer**. Essa etapa consiste em ordenar as macros de forma que toda macro seja definida antes de seus usos, produzindo um grafo G_m que modela o relacionamento entre as macros.

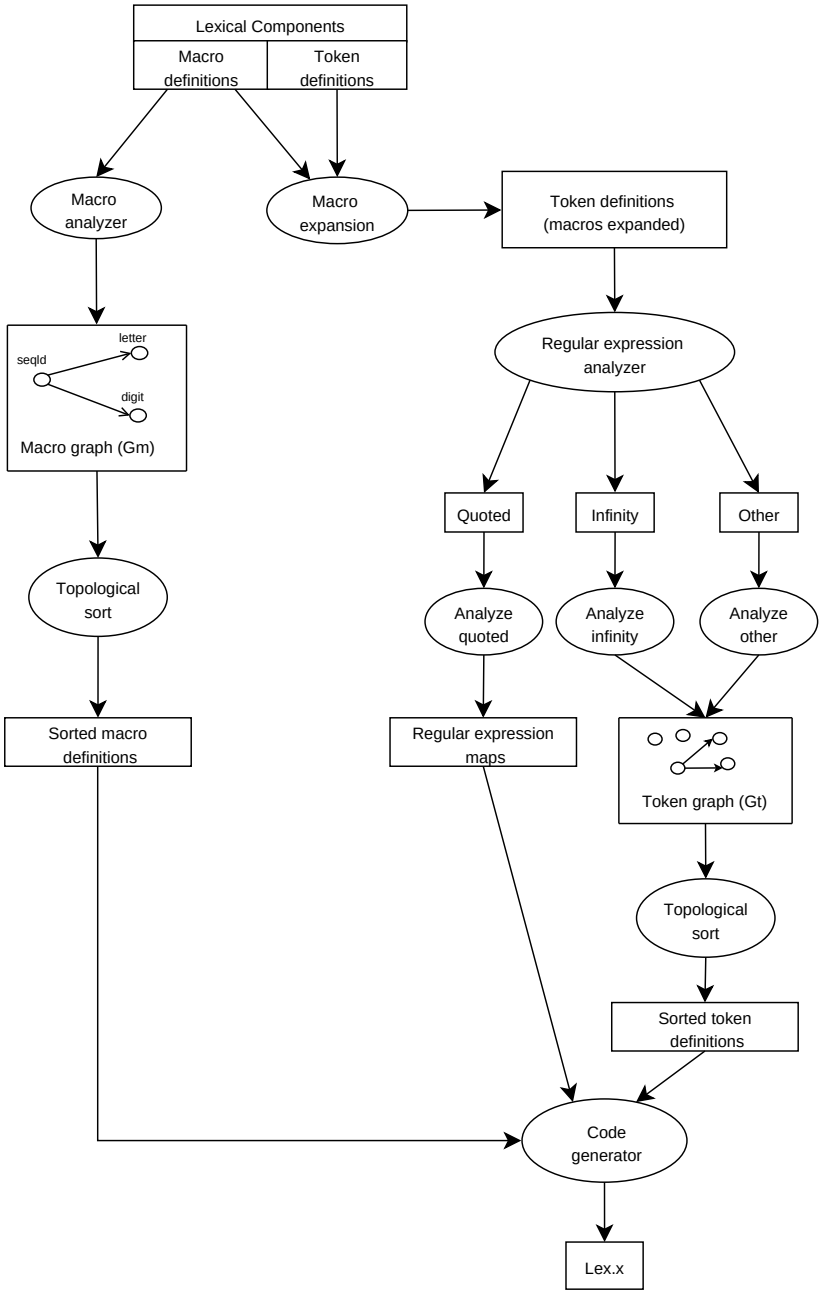

Figura 4.6: Diagrama ilustrando as etapas de tradução da descrição léxica em *Notus* de uma linguagem para um arquivo de entrada para o *Alex*.54

As macros e os *tokens* da especificação são tratados pela etapa **Macro expansion**,

que consiste em expandir as macros presentes na definição léxica, substituindo seus usos pelas expressões regulares que as definem. Essa expansão é necessária para que a etapa **Regular expression analyzer** defina a ordem de geração das expressões no arquivo fonte gerado para o *Alex*. Adicionalmente, a etapa **Regular expression analyzer** particiona as expressões regulares em subconjuntos a serem analisados separadamente pelos algoritmos **Analyze quoted**, **Analyze infinity**, e **Analyze other**, detalhados a seguir. O primeiro algoritmo é responsável por otimizar o analisador léxico gerado, quando possível, evitando a geração de estados no autômato de reconhecimento de *tokens* para palavras-chave da linguagem especificada. Os outros dois são responsáveis pela criação de um grafo G_m representando a relação existente entre as expressões regulares dos *tokens* da linguagem.

Finalmente, o algoritmo de ordenação topológica [Sedgewick, 1990] é executado sobre os grafos G_m e G_t obtendo-se a ordem na qual as macros e os *tokens* devem ser gerados pela etapa **Code generator** no arquivo-fonte *Lex.x*.

As seções seguintes detalham as etapas, seus algoritmos e as informações que produzem.

4.1.1 Ordenação das Macros

É importante observar que a linguagem Notus foi projetada visando a construção de descrições formais modulares e incrementais. Dessa maneira, o projetista escreve um novo módulo sem se preocupar com sua integração aos demais. Exigir que o projetista defina a ordem das macros sempre que um novo módulo seja construído ou sempre que ocorra alguma alteração violaria este objetivo de *Notus*. Assim o compilador *Notus* analisa as macros e define sua ordem no arquivo *Lex.x*. No exemplo da Figura 4.5, a macro *seqld* deve aparecer após as definições das macros *letra* e *digito*, uma vez que ambas são utilizadas na definição de *seqld*. Além disso, o compilador de *Notus* também verifica se há ou não recursividade nas definições de macros, o que não é permitido na linguagem *Notus*, segundo [Tirelo e Bigonha, 2006].

A ordenação de macros é modelada por meio de um grafo $G_m = (V_m, E_m)$ em que cada vértice $v \in V_m$ representa uma definição de macro, e cada aresta $e = (u, v) \in E_m$, onde $u, v \in V_m$, representa uma dependência entre macros e indica que a macro correspondente a u depende da macro correspondente a v. Um algoritmo de ordenação topológica [Sedgewick, 1990] para grafos acíclicos dirigidos é executado sobre o grafo G_m para determinar uma ordem válida das macros. Dessa maneira, no grafo da Figura 4.7, correspondente ao exemplo da Figura 4.5, existem arestas direcionadas do vértice que representa `seqld` para os vértices que representam `letter` e `digit`. O rótulo *uses* é utilizando apenas para tornar a figura mais legível, e não existe no grafo G_m.

O arquivo *Lex.x* gerado lista as definições das macros de acordo com o resultado da ordenação topológica do grafo G_m.

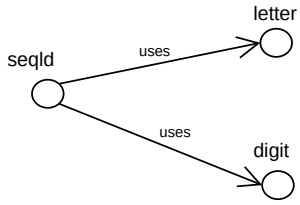

Figura 4.7: Grafo G_m para as macros do exemplo da Figura 4.5

4.1.2 Expansão das Macros

O compilador de *Notus* realiza a expansão de macros, que consiste em substituir a ocorrência de uma macro pela expressão regular que a define. Em *Notus* é possível que a definição de uma macro referencie outras. Para tratar a expansão de macros nesses casos, o compilador de *Notus* implementa um algoritmo de ponto fixo. Esse algoritmo expande iterativamente as macros que definem os *tokens* e as próprias macros da linguagem até que nenhuma nova macro seja expandida. A Listagem 4.1 mostra os resultados intermediários do algoritmo para a expansão de macros da especificação da Listagem 3.8 (veja Página 39), excluindo por simplicidade a declaração dos *tokens* correspondentes às palavras-chave *if*, *then*, *else*, *while* e *do*.

```
1  // Passo 1:
2  id=seqId
3  num=digito+
4  seqId=letra(letra|digito)*
5  letra=[A–Z]|[a–z]
6  digito=[0–9]
7  ---------------------------------------------------------------
8  // Passo 2:
9  id=(letra(letra|digito)*)
10 num=([0–9])+
11 seqId=([A–Z]|[a–z])(([A–Z]|[a–z])|([0–9]))*
12 letra=[A–Z]|[a–z]
13 digito=[0–9]
14 ---------------------------------------------------------------
15 // Passo 3:
16 id=(([A–Z]|[a–z])(([A–Z]|[a–z])|([0–9]))*)
17 num=([0–9])+
```

```
18 seqId=([A–Z]|[a–z])(([A–Z]|[a–z])|([0−9]))*
19 letra=[A–Z]|[a–z]
20 digito=[0−9]
21 ---------------------------------------------------------------
22 // Passo 4:
23 id=(([A–Z]|[a–z])(([A–Z]|[a–z])|([0−9]))*)
24 num=([0−9])+
25 seqId=([A–Z]|[a–z])(([A–Z]|[a–z])|([0−9]))*
26 letra=[A–Z]|[a–z]
27 digito=[0−9]
```

Listagem 4.1: Resultados intermediários do algoritmo para expansão de macros

4.1.3 Análise das Expressões Regulares dos Tokens

Após o tratamento das macros, a análise dos componentes léxicos da definição passa a operar sobre as definições dos *tokens* da linguagem. Geralmente, quando mais de uma expressão regular for capaz de denotar um mesmo *string* de entrada, os analisadores léxicos resolvem esse conflito optando pela expressão regular capaz de denotar o maior *string*. Caso mais de uma expressão reconheça *strings* de mesmo tamanho, o analisador léxico prioriza a expressão regular que aparece antes no arquivo de definições de expressões regulares.

Assim, as definições léxicas presentes nos vários módulos devem ser processadas para definir a ordem das expressões no arquivo fonte gerado para o *Alex*, *Lex.x*. Para a definição da ordem, cada expressão regular presente em um dos vários módulos deve ser comparada às demais da seguinte forma: sejam R_1 e R_2 expressões regulares, e $L(R_1)$ e $L(R_2)$, as respectivas linguagens denotadas. Para cada par de expressões (R_1, R_2) pertencentes à definição, verifica-se se $L(R_1)$ está contida em $L(R_2)$. Se isso ocorrer, a expressão R_1 deve aparecer antes de R_2 na definição do analisador léxico gerado. Na Figura 4.5 as expressões regulares que definem as palavras-chave "`if`", "`then`" e "`else`" estão contidas na expressão regular `letter( letter | digit )*`, que define o *token* `seqId`, e por isso aparecem antes dessa no arquivo *Lex.x* gerado.

O algoritmo para verificar se $L(R_1)$ está contida em $L(R_2)$ é realizado pela operação de diferença dos autômatos gerados por R_1 e R_2. A linguagem $L(R_1)$ está contida em $L(R_2)$ se e somente se $L(R_1) - L(R_2) = \oslash$. Esta operação é feita utilizando-se autômatos finitos determinísticos (AFD) que reconhecem $L(R_1)$ e $L(R_2)$. Na prática a diferença entre as linguagens é obtida como mostrado no procedimento CONTAINED[4],

[4]Os pseudo-códigos apresentados neste texto utilizam o pacote para *latex clrscode* [Cormen, 2003], em que o símbolo ▷ indica comentário de linha.

onde M_1 e M_2 são AFD's gerados a partir de R_1 e R_2 [Sipser, 1996]. A operação de complemento de linguagens por meio de autômatos possui complexidade linear no número de estados do AFD, e a interseção possui complexidade quadrática no número de estados dos AFDs envolvidos.

```
CONTAINED(R1, R2)
 1  M1 ← TOAUTOMATON(R1);
 2  M2 ← TOAUTOMATON(R2);
 3  M2_Complement ← COMPLEMENT(M2);
 4  res ← INTERSECTION(M1, M2_Complement);
 5  if EMPTY(res)
 6     then ▷ L(R1) ⊂ L(R2)
 7          return TRUE;
 8     else ▷ L(R1) ⊄ L(R2)
 9          M1_complement← COMPLEMENT(M1);
10          res2 ← INTERSECTION(M1_complement,M2);
11          if ¬ EMPTY(res2)
12             then ERROR("The token definitions given by regular
13                  expressions "+R1+" and " + R2 +" are ambiguous");
14          return FALSE;
```

O compilador de *Notus* utiliza a biblioteca em Java *automaton*[5] para manipulação de expressões regulares e autômatos. Os métodos TOAUTOMATON, COMPLEMENT, INTERSECTION e EMPTY usados no procedimento CONTAINED fazem parte desta biblioteca.

Com as macros expandidas, o compilador de *Notus* aplica o procedimento CONTAINED. Para evitar que cada expressão regular tenha que ser comparada com todas as demais, somente alguns tipos de verificações são realizados com base nas propriedades das linguagens regulares. Inicialmente, o conjunto das expressões regulares que definem os *tokens* da linguagem é particionado em 3 subconjuntos:

1. *QUOTED*: expressões regulares que representam *tokens* da linguagem identificados por *strings* entre aspas

2. *INFINITY*: expressões regulares que denotam linguagens infinitas, detectadas pela presença dos operadores + e $*$ na expressão regular[6];

[5]Disponível no endereço `http://www.brics.dk/~amoeller/automaton/`

[6]As linguagens regulares finitas denotadas pelas expressões $\emptyset*$ e $\lambda*$ não são permitidas em *Notus*, o que permite que este teste seja suficiente.

3. *OTHER*: demais expressões regulares que não pertencem a nenhum dos grupos anteriores; as linguagens denotadas pelas expressões regulares deste conjunto são, portanto, finitas.

Estes três subconjuntos foram comparados entre si para identificação de expressões regulares que denotam linguagens contidas em outras, como mostra o procedimento ANALYZEREGEXP. No entanto, é possível observar nos procedimentos que analisam as expressões desses conjuntos, que algumas comparações foram evitadas. Esta melhoria é importante, pois o procedimento CONTAINED, responsável por esta verificação, envolve operações como complemento e interseção, que requerem autômatos determinísticos, e a operação de transformar um autômato não-determinístico em determinístico possui complexidade de tempo exponencial no número de estados [Sudkamp, 1997].

```
ANALYZEREGEXP(QUOTED,OTHER,INFINITY)
1  for each q ∈ QUOTED
2       do ANALYZEQUOTED(q,OTHER,INFINITY)
3  for each o ∈ OTHER
4       do ANALYZEOTHER(o,QUOTED,OTHER,INFINITY)
5  for each i ∈ INFINITY
6       do ANALYZEINFINITY(i,INFINITY)
```

O procedimento ANALYZEREGEXP é importante para determinar a ordem em que as definições dos *tokens* devem ser geradas em *Alex*. Adicionalmente, um analisador léxico otimizado, com um número menor de estados, é gerado utilizando-se as informações coletadas no procedimento ANALYZEQUOTED.

O procedimento ANALYZEQUOTED verifica a possibilidade de uma linguagem denotada por uma expressão regular $R_1 \in QUOTED$ estar contida na linguagem denotada por $R_2 \in OTHER \cup INFINITY$. O procedimento RUN pertencente à biblioteca *automaton* é responsável por verificar se um autômato reconhece o *string* que compõe uma expressão regular. Sendo este teste verdadeiro, tem-se que a linguagem denotada por R_1 está contida em R_2, e R_1 é inserida em uma lista auxiliar, *containedRegExps*, associada a R_2.

```
ANALYZEQUOTED(R1,OTHER,INFINITY)
1  for each R2 ∈ OTHER ∪ INFINITY
2       do M2 ← TOAUTOMATON(R2)
3          if RUN(M2,R1)
4             then ▷ L(R1) ⊂ L(R2)
5                  containedRegExps[R2] ← containedRegExps[R2] + R1
```

Para cada expressão regular com uma lista associada composta por elementos $s \in QUOTED$, é criado um mapeamento destes elementos em um *closure* contendo o nome do *token* definida por s, e a função de tratamento de lexema para este *token*, caso exista. Desta maneira, para o código da Figura 4.5, a lista associada à expressão regular `seqld` $\in INFINITY$, que define identificadores, é ("if", "*then*", "*else*", "*while*", "*do*"). O mapeamento criado para `seqld` é mostrado na Figura 4.8. Note que os *closures* criados são compostos apenas pelos nomes dos *tokens*, já que estes não possuem funções de lexemas associadas.

$$
\begin{array}{lll}
\{ \text{ “if”} & \mapsto & \text{if,} \\
\text{ “then”} & \mapsto & \text{then,} \\
\text{ “else”} & \mapsto & \text{else,} \\
\text{ “while”} & \mapsto & \text{while,} \\
\text{ “do”} & \mapsto & \text{do } \}
\end{array}
$$

Figura 4.8: Mapeamento criado para a expressão regular `seqld`

Para as expressões regulares que possuem mapeamento associado, gera-se uma ação semântica que verifica se o *string* casado está associado no mapeamento; se estiver, o resultado é o *token* correspondente, com lexema dado pela aplicação da função de tratamento ao *string* casado; se não estiver, o resultado é a própria definição do *token*. Para o exemplo da Figura 4.5, o casamento do *string if* teria como resultado if e o casamento de um identificador qualquer "x" teria como resultado id. Assim, esta otimização permite que as definições de *tokens* referentes às palavras-chave da linguagem sejam eliminadas, com a conseqüente diminuição do tamanho das tabelas de autômatos geradas pelo *Alex*.

Os procedimentos analyzeOther e analyzeInfinity são utilizados para a criação de um grafo $G_t = (V_t, E_t)$, onde cada $v \in V_t$ corresponde a uma expressão regular de definição de *token*, e para cada $u, v \in V_t$, tem-se que $(u, v) \in E_t$ se, considerando que u representa a expressão regular R_1 e v, a expressão regular R_2, $L(R_1) \subset L(R_2)$. A partir deste grafo, executa-se o algoritmo de ordenação topológica, que informa uma ordem na qual os *tokens* devem ser gerados.

O procedimento analyzeOther verifica a possibilidade de uma linguagem denotada por uma expressão regular $R_1 \in OTHER$ estar contida na linguagem denotada por $R_2 \in INFINITY \cup QUOTED \cup OTHER$. Caso isso aconteça, cria-se uma aresta de R_1 para R_2, adicionando-se R_2 na lista de adjacência de R_1 (linha 4).

```
analyzeOther(R1,QUOTED,OTHER,INFINITY,Gt)
1  for each R2 ∈ INFINITY ∪ QUOTED ∪ OTHER
2       do if contained(R1, R2)
3            then ▷ L(R1) ⊂ L(R2)
4                 adjacencieList[Gt,R1] ← adjacencieList[Gt,R1] + R2
```

O procedimento analyzeInfinity verifica a possibilidade de uma linguagem denotada por uma expressão regular $R_1 \in INFINITY$ estar contida na linguagem denotada por $R_2 \in INFINITY$. Caso isso aconteça, cria-se uma aresta de R_1 para R_2, adicionando-se R_2 na lista de adjacência de R_1 (linha 4).

```
analyzeInfinity(R1, INFINITY, Gt)
1  for each R2 ∈ INFINITY
2       do if contained(R1, R2)
3            then ▷ L(R1) ⊂ L(R2)
4                 adjacencieList[Gt,R1] ← adjacencieList[Gt,R1] + R2
```

Para efeitos de comparação, arquivos *Lex.x* foram gerados com e sem a otimização de eliminação de *tokens*[7] contidos em outros, como exibido na Figura 4.9. Por motivos de simplificação, não foi mostrado nesta figura o mapeamento criado para o *token seqld* mostrado na Figura 4.8. Com a otimização, o número de estados do autômato finito gerado pelo *Alex* foi 4, e, sem a otimização, 23. O tamanho das tabelas para análise léxica, criadas pelo *Alex*, também variou nas duas estratégias. O analisador léxico gerado com a otimização manipula tabelas com 417 entradas e sem a otimização com 1568 entradas. É importante observar que esses números variam de acordo com a linguagem que está sendo definida, principalmente com o número de palavras-chave que a linguagem possui.

4.1.4 Geração de Código

As regras de tradução das construções de *Notus* para *Alex* são descritas em semântica denotacional nas Figuras 4.10 e 4.11. As regras de tradução descrevem o processo de geração de código das construções da especificação léxica em *Notus* discutida na Seção 3.5, para as construções correspondentes de uma especificação léxica em *Alex*.

As seções de definição dos domínios sintáticos e semânticos da Figura 4.10 declaram os domínios usados na especificação. A seção de sintaxe abstrata formaliza as construções de *Notus* para a especificação léxica discutida na Seção 3.5. A especificação léxica pode conter declarações de *tokens*, macros, por meio da cláusula *element*, e expressões regulares a serem ignoradas por meio da cláusula *ignore*.

[7] Na verdade o que se analisa são as expressões regulares que definem os *tokens* da linguagem.

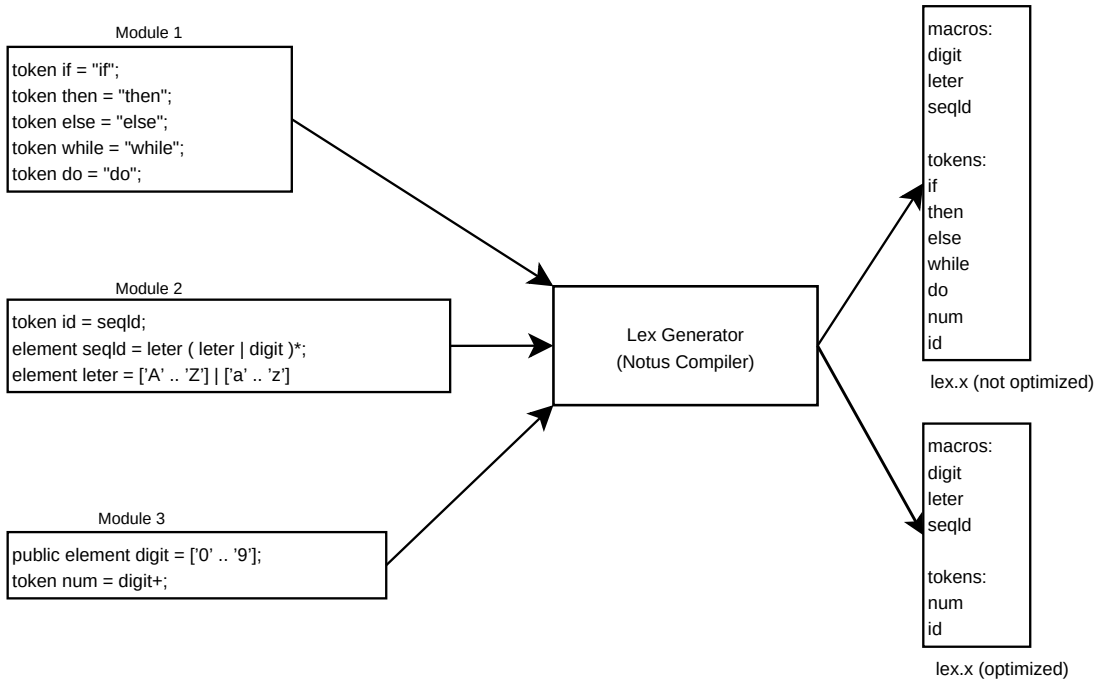

Figura 4.9: Composição da descrição léxica em arquivos de entrada (com e sem a otimização proposta) para o programa *Alex*

Expressões regulares em *Notus* podem ser compostas por caracteres, *strings*, identificadores de macros ou por operadores. Os operadores binários existentes são o de união, representado por '|', e o de concatenação, representado por simples justaposição de expressões. Os unários são o operador de complemento (ˆ), que representa o complemento da expressão regular que o sucede, o operador de *Kleene star* (∗), o operador de *Kleene plus* (+) e o operador de opção (?) que indicam, respectivamente, zero ou mais repetições, uma ou mais repetições, e no máximo uma ocorrência da expressão regular que os antecedem. A expressão regular '.' reconhece qualquer caractere exceto a quebra de linha ("`\n`").

A função de compilação de declaração, cujo tipo é determinado na Figura 4.10, recebe uma declaração e um estado e produz o código referente a essa declaração na notação aceita por *Alex*. O mesmo acontece com a função de compilação de expressões regulares que, dados uma expressão e um estado, gera código para essa expressão seguindo o especificado pelo *Alex* para expressões regulares. O estado é modelado por uma função que associa a cada identificador um valor lógico indicando se o identificador corresponde ou não a uma macro. Essa distinção é necessária para a geração de código para a macro, já que de acordo com a especificação do *Alex*, todo identificador de macro deve ser precedido pelo símbolo '@'.

As equações semânticas da Figura 4.11 traduzem cada construção de especificação léxica de *Notus*, para as construções equivalentes em *Alex*. O operador "?", que aparece

Domínio Sintático:

$$
\begin{array}{lcl}
Bool & = & \{true, false\} \\
Ide & = & \{I | I \text{ é um identificador}\} \\
Exp & = & \{E | E \text{ é uma expressão regular}\} \\
Dec & = & \{D | D \text{ é uma declaração}\} \\
String & = & \{S | S \text{ é um string entre aspas duplas}\}
\end{array}
$$

Sintaxe Abstrata:

$$
\begin{array}{lcl}
D & ::= & \textbf{token } I_1 \ : \ I_2? \ = \ E_1 \textbf{ is } E_2 \\
 & | & \textbf{token } I_1 \ : \ I_2? \ = \ E_1 \\
 & | & \textbf{element } I \ = \ E \\
 & | & \textbf{ignore } E \\
E & ::= & (E_1 | E_2) \mid \text{“ ~ ”} E \mid E \text{ “*” } \mid E \text{ “+” } \mid E \text{ “?”} \\
 & | & \text{“(”} \ E \ \text{“)”} \mid \text{“[”} \ Char1 \text{ - } Char2 \text{ “]”} \mid S \mid I \mid \text{“.”}
\end{array}
$$

Domínios Semânticos:

$\textbf{State} = [\textbf{Ide} \mapsto \textbf{Bool}]$
$s \in \textbf{State}$

Funções Semânticas de Compilação:

- Compilação de Declaração
  $\mathcal{KD} : \textbf{Dec} \mapsto \textbf{State} \mapsto \textbf{IO}$
- Compilação de Expressão
  $\mathcal{KE} : \textbf{Exp} \mapsto \textbf{State} \mapsto \textbf{IO}$

Funções Auxiliares:

$print : \textbf{String} \mapsto \textbf{IO}$

Figura 4.10: Regras de tradução de *Notus* para a especificação léxica

após o identificador representando o nome de um domínio sintático, indica que esses elementos são opcionais. Para o *Alex*, um *token* é definido escrevendo-se sua expressão regular (R) seguida por código *Haskell* entre {}, que corresponde à ação a ser executada quando um *string* denotado por R é reconhecido. O *Alex* requer que todas as ações possuam o mesmo tipo. O compilador de *Notus* gera ações do tipo $String \rightarrow Token$, onde o parâmetro *String*, representado por s nas ações geradas, corresponde ao lexema reconhecido, e o tipo *Token*, um *datatype* definido em *Haskell*, possui um construtor de tipos para cada *token* declarado pela linguagem especificada. Se uma declaração de *token* define uma função para o seu lexema pelo uso da cláusula **is**, o compilador de *Notus* gera uma ação que aplica essa função a s.

A equação semântica para a declaração de macros, além de gerar o símbolo @ antes

Equações Semânticas Para Geração de Código:

$$
\begin{array}{lcl}
\mathcal{KD}[\![\textbf{token } I_1 : I_2? = E_1 \textbf{ is } E_2]\!]s_0 & = & \mathcal{KE}[\![E_1]\!]s_0 \bullet print(\text{“\{\textbackslash}s \rightarrow \text{”})\bullet \\
 & & \mathcal{KE}[\![I_1]\!]s_0 \bullet print(\text{“(”})\bullet \\
 & & \mathcal{KE}[\![E_2]\!]s_0 \bullet print(\text{“ }s)\}\text{”}) \\
 & & \\
\mathcal{KD}[\![\textbf{token } I_1 : I_2? = E_1]\!]s_0 & = & \mathcal{KE}[\![E_1]\!]s_0 \bullet print(\text{“\{\textbackslash}s \rightarrow \text{”})\bullet \\
 & & \mathcal{KE}[\![I_1]\!]s_0 \bullet print(\text{“\}”}) \\
 & & \\
\mathcal{KD}[\![\textbf{element } I \ = \ E]\!]s_0 & = & print(\text{“@”}) \bullet \mathcal{KE}[\![I]\!]s_0\bullet \\
 & & print(\text{“ = ”}) \bullet \mathcal{KE}[\![E]\!]s_0 \\
 & & \\
\mathcal{KD}[\![\textbf{ignore } E]\!]s_0 & = & \mathcal{KE}[\![E]\!]s_0 \bullet print(\text{“;”}) \\
 & & \\
\mathcal{KE}[\![(E_1|E_2)]\!]s_0 & = & \mathcal{KE}[\![E_1]\!]s_0 \bullet print(\text{“|”}) \bullet \mathcal{KE}[\![E_2]\!]s_0 \\
 & & \\
\mathcal{KE}[\![\text{“ \^{} ”}E]\!]s_0 & = & print(\text{“\^{}”}) \bullet \mathcal{KE}[\![E]\!]s_0 \\
 & & \\
\mathcal{KE}[\![E \text{ “*”}]\!]s_0 & = & \mathcal{KE}[\![E]\!]s_0 \bullet print(\text{“*”}) \\
 & & \\
\mathcal{KE}[\![E \text{ “+”}]\!]s_0 & = & \mathcal{KE}[\![E]\!]s_0 \bullet print(\text{“+”}) \\
 & & \\
\mathcal{KE}[\![E \text{ “?”}]\!]s_0 & = & \mathcal{KE}[\![E]\!]s_0 \bullet print(\text{“?”}) \\
 & & \\
\mathcal{KE}[\![\text{“(”} \ E \ \text{“)”}]\!]s_0 & = & print(\text{“(”}) \bullet \mathcal{KE}[\![E]\!]s_0\bullet \\
 & & print(\text{“)”}) \\
 & & \\
\mathcal{KE}[\![\text{“[”} \ Char_1 \text{ - } Char_2 \ \text{“]”}]\!]s_0 & = & print(\text{“[”}Char_1 \ \text{“}-\text{”} \ Char_2 \ \text{“]”}) \\
 & & \\
\mathcal{KE}[\![S]\!]s_0 & = & print(S) \\
 & & \\
\mathcal{KE}[\![I]\!]s_0 & = & s_0 \ I \rightarrow print(\text{“@ }I\text{”}) \ , (print(\text{“}I\text{”}) \ ) \\
\mathcal{KE}[\![\text{“.”}]\!]s_0 & = & print(\text{“.”})
\end{array}
$$

Figura 4.11: Regras de tradução de *Notus* para a especificação léxica.

do identificador que o define, possui o efeito colateral de associar na função *State* o identificador da macro ao valor lógico *true*, indicando que esse identificador é macro. A definição de uma macro em *Alex* segue o formato:

$$@identificador\_macro = expressão\_regular$$

Os operadores que podem aparecer na definição de uma expressão regular em *Notus* correspondem aos usados pelo *Alex*. Dessa forma, a tradução de expressões regulares

é direta. Note que, na equação semântica para identificadores, verifica-se se o identificador é macro para que o símbolo @ seja gerado somente quando necessário. Durante a fase de análise semântica, associa-se a cada identificador um valor lógico indicando se ele corresponde ou não a uma macro. Esse mapeamento é armazenado em s_0.

O código gerado, mostrado nas Listagens 4.3 e 4.2, é submetido ao *Alex* para a obtenção de um analisador léxico para a linguagem especificada no exemplo da Figura 4.5. O módulo *Tree* da Listagem 4.2 é importado por toda definição em *Alex* gerada por *Notus*. Esse módulo contém a definição de uma árvore binária e uma função de pesquisa nesta árvore.

```
1
2  module Tree where
3  data Tree a b = Leaf (a,b)
4           | Node (a,b) (Tree a b) (Tree a b)
5           | Null
6      deriving Eq
7
8  lookupT :: Ord a => Tree a Token -> a -> Token -> Token
9  lookupT (Leaf (a,b)) s rootToken
10   | s == a = b
11   | otherwise = rootToken
12 lookupT (Node (n,t) l r) s rootToken
13   | n == s  = t
14   | s > n   = lookupT r s rootToken
15   | s < n   = lookupT l s rootToken
16 lookupT Null _ rootToken = rootToken
```

Listagem 4.2: Código do módulo *Tree*.

4.2 Compilação da Especificação Sintática

O compilador de *Notus* gera um arquivo de entrada para o *Happy* com base na especificação sintática de uma linguagem. Esse arquivo é composto pela definição da gramática da linguagem e por declarações de *datatypes* que representam nós da árvore de sintaxe abstrata de um programa dessa linguagem. Essa seção descreve o processo de compilação da especificação sintática de uma linguagem até a obtenção de todas informações necessárias para a construção do arquivo de entrada do *Happy*: *Syn.y* da Figura 4.2.

A sintaxe de uma linguagem deve ser especificada, em *Notus*, por uma gramática livre de contexto $LALR(1)$ [Tirelo e Bigonha, 2006]. As regras de produção da gramática são definidas usando a notação *BNF*, estendida com símbolos especiais para indicar repetição. Uma regra de produção R tem a forma: A::= $\alpha_1, \alpha_2, ..., \alpha_n$, onde A é uma

```
1  {
2          module Lexico where
3           import Tree
4  }
5
6  %wrapper "basic"
7
8  @digito = [0-9]
9  @letra = [A-Z]|[a-z]
10 @seqId = @letra( @letra|@digito )*
11
12 tokens :-
13
14 @seqId           { \s->lookupT idTree s (T__id)}
15 @digito+                  { \s->(T__num)}
16
17 {
18 -- The token type:
19 data Token
20         = T__if
21         | T__then
22         | T__else
23         | T__while
24         | T__do
25         | T__id
26         | T__num
27  deriving Show
28
29 idTree = (Node ("if",(T__if))
30           (Node ("else",(T__else))(Leaf ("do",(T__do))) Null )
31           (Node ("then",(T__then)) Null (Leaf ("while",(T__while)))))
32
33 }
```

Listagem 4.3: Código gerado para o exemplo da Figura 4.5.

variável, cada α_i é um constituinte de regra, e $n \geq 0$. O constituinte de regra é um identificador de *token* ou variável.

Para facilitar a escrita de regras de produção, *Notus* possui símbolos que representam repetição de constituintes de regras. Esses símbolos são listados a seguir:

- $\alpha*$, indica 0 ou mais ocorrências de α.
- $\alpha+$, indica 1 ou mais ocorrências de α.
- $\alpha*$-t, indica 0 ou mais ocorrências de α, separadas pelo *token* t.
- $\alpha+$-t, indica 1 ou mais ocorrências de α, separadas pelo *token* t.

A regra de produção com a forma $A \rightarrow \lambda$ pode ser especificada usando uma das duas regras a seguir:

```
A ::=
A ::= empty
```

Geradores de analisadores sintáticos tradicionais, como o *Happy*, não possuem construções específicas para descrever repetição. Assim, é preciso que as regras de produção contendo esses símbolos sejam traduzidas para a sintaxe *BNF*. A tradução das construções gramaticais de *Notus* para *BNF* é detalhada na Seção 4.2.1.

A sintaxe abstrata de uma linguagem em *Notus* pode ser automaticamente gerada pelo compilador de *Notus*, ou o projetista pode adicionar a cada regra de produção uma regra abstrata alternativa. O código da Listagem 4.4 exemplifica a criação de regras abstratas para produções de uma gramática.

```
public exp : Exp ::= exp "+" term : [ "add" exp term ] // rule 1
                   | exp "-" term : [ "sub" exp term ] // rule 2
                   | term : term;                      // rule 3
term : Exp ::= term "*" factor : [ "mul" term factor ] // rule 4
             | term "/" factor : [ "div" term factor ] // rule 5
             | factor : factor;                        // rule 6
factor : Exp ::=  id : id                              // rule 7
               | num : num                             // rule 8
               | "(" exp ")" : exp ;                   // rule 9
```

Listagem 4.4: Exemplo de definição de sintaxe abstrata

A definição de regras abstratas pode ser feita de duas maneiras. A primeira técnica consiste em adicionar à regra de produção uma construção da forma:

$$A \rightarrow a_1 a_2 \ldots a_n : [b_1 b_2 \ldots b_m]$$

onde cada b_i é um dos a_j's ou um novo símbolo entre aspas duplas. Se A possui domínio d e cada b_i possui domínio d_i, a gramática abstrata contém a regra $d \rightarrow d_1 d_2 \ldots d_m$. As regras 1, 2, 4 e 5 na Listagem 4.4 são exemplos dessa técnica.

A segunda técnica é a definição de uma regra para ignorar a indireção de uma produção, ou seja, cria-se uma ligação direta entre a variável sendo definida e o símbolo da indireção. A construção para ignorar a indireção possui a seguinte forma:

$$A \rightarrow a_1 a_2 \ldots a_n : a_i$$

onde a_i é um símbolo pertencente a $\{a_1, a_2, \ldots, a_n\}$. Nesse caso, considerando que A possua domínio d, e a_i possua domínio d_i, então a gramática abstrata contém regras $d \rightarrow b_1, b_2, \ldots b_m$, para cada regra $d_i \rightarrow b_1, b_2, \ldots b_m$, ou seja, as produções da gramática abstrata de a_i são incorporadas às produções abstratas de A. As Regras 7, 8 e 9 na Listagem 4.4 são exemplos dessa técnica. As Regras 3 e 6 não criam regras de sintaxe abstrata alternativas, sendo o compilador responsável por criá-las. A Seção 4.2.1 detalha como o compilador de *Notus* gera a sintaxe abstrata de uma linguagem.

4.2.1 Visão Geral

A Figura 4.12 ilustra o fluxo de dados e as etapas da tradução dos componentes sintáticos da especificação de uma linguagem em Notus para um arquivo de entrada para o *Happy*. Cada etapa da tradução realizada pelo compilador de *Notus* é representado por uma caixa oval, e os dados de entrada e gerados pelos compilador são representados por retângulos.

Inicialmente o compilador de *Notus* analisa a gramática especificada, na etapa **Grammar analyzer**, gerando uma gramática concreta, em conformidade com a notação *BNF*, e uma gramática abstrata. As Seções 4.2.2 e 4.2.3 detalham como esse processo é realizado.

A gramática concreta resultante é simplificada, na etapa **Grammar simplification**, por meio da eliminação de variáveis que não geram *strings* e de símbolos não-alcançáveis. A etapa de simplificação da gramática é detalhada na Seção 4.2.4.

A gramática abstrata resultante da etapa **Grammar analyzer** é usada para a geração de *datatypes* em *Haskell* pela etapa **Datatypes generation**, detalhada na Seção 4.2.5. Os *datatypes* gerados por essa etapa compõem o arquivo *Syn.y*.

A gramática concreta simplificada e os *datatypes* são usados pela etapa **Semantic actions generation** para a geração das ações semânticas das produções da gramática gerada a ser processada pelo *Happy*. Essa etapa, detalhada na Seção 4.2.6, é responsável por associar a cada regra de produção da gramática uma ação semântica.

A gramática com ações semânticas associadas a cada uma de suas produções é submetida à etapa **Code generator**, detalhada na Seção 4.2.7, para a geração da sintaxe da linguagem em Happy.

Os *datatypes*, gerados na etapa **Datatypes generation**, e a definição sintática em Happy, gerada na etapa **Code generator**, são gravados no arquivo *Syn.y*.

As seções seguintes detalham as etapas, seus algoritmos e as informações que produzem.

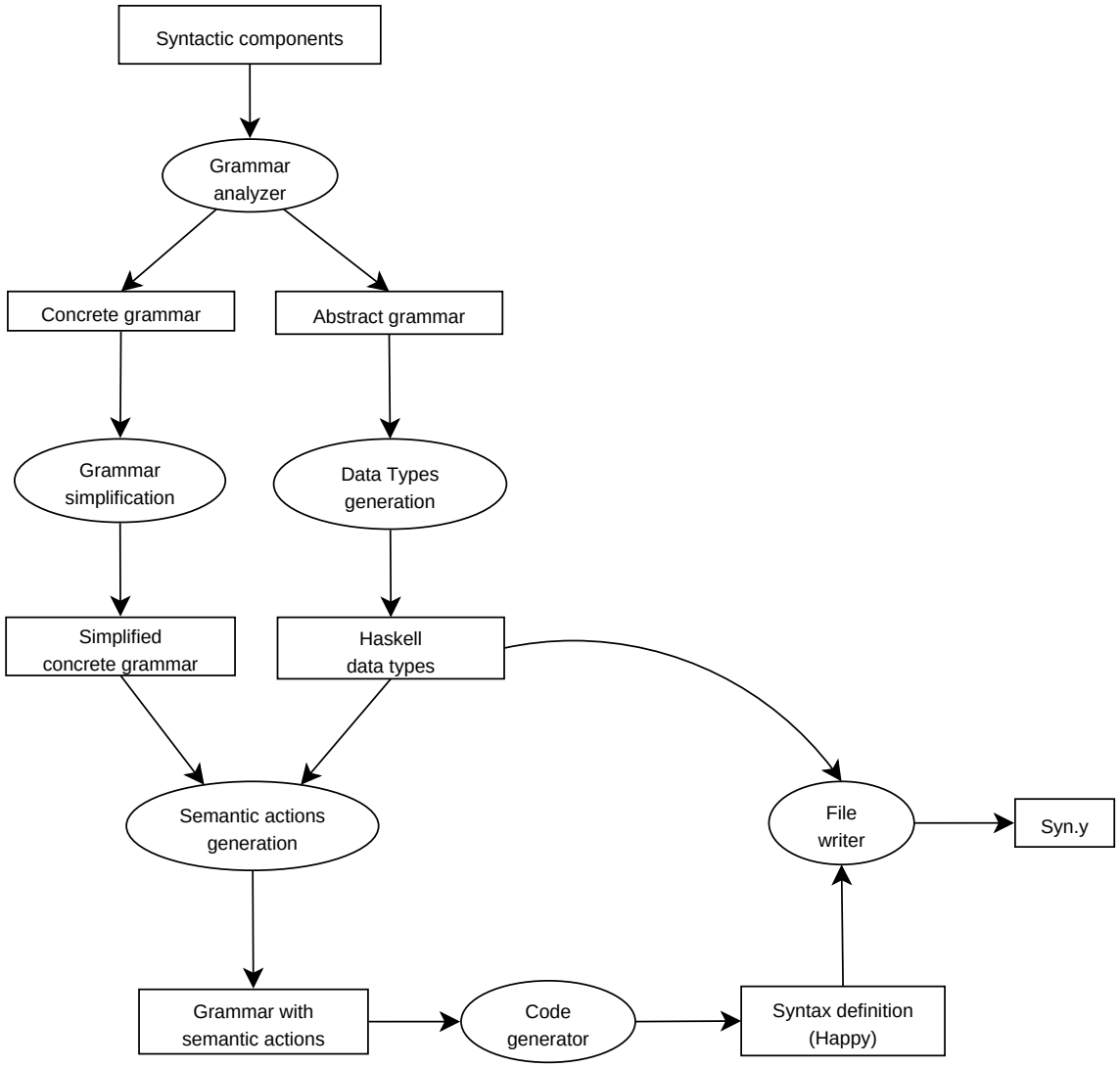

Figura 4.12: Fluxo de dados e etapas de tradução da descrição sintática de uma linguagem em um arquivo de entrada para o *Happy*, *Syn.y*.

4.2.2 Transformação da Gramática Concreta

O processo de análise semântica realizado sobre as regras de produção de uma gramática em *Notus* consiste em obter, a partir da gramática definida, uma nova gramática em conformidade com a notação *BNF* processável pelo *Happy*. Seja $G = (V, T, R, S)$ uma gramática definida em *Notus* para alguma linguagem arbitrária L onde:

- V é o conjunto de todas as variáveis declaradas nos módulos da especificação de L;
- T é o conjunto dos terminais de L, composto pelos *tokens* declarados nos módulos da definição de L e pelos símbolos entre aspas duplas (*tokens* anônimos) presentes nas regras de produção da definição da gramática de L;

- R é o conjunto de regras de produção, composto pelas regras definidas nos módulos da especificação de L;

- $S \in V$ é o símbolo inicial da gramática, definido no módulo principal de L (veja Seção 3.6)

O algoritmo transformGrammar é aplicado a uma gramática $G = \{V, T, R, S\}$ em *Notus*, obtendo-se uma nova gramática $G' = \{V', T, R', S\}$, cujas regras de produção estão em conformidade com a notação *BNF* padrão, tal que $L(G) = L(G')$, isto é, G e G' geram a mesma linguagem.

```
transformGrammar(G)
1  for each r ∈ R[G]
2       do for each rc ∈ r
3            do ct ← analyzeRuleConstituent(rc)
4               nr ← nr + nrc[t]
5               R'[G'] ← R'[G'] ∪ rSet[ct]
6               V'[G'] ← V'[G'] ∪ vSet[ct]
7          R'[G'] ← nr ∪ R'[G']
8  return G'
```

O algoritmo transformGrammar percorre cada regra r de produção da gramática, e chama, para cada constituinte rc de uma regra, o algoritmo analyzeRuleConstituent, que possui uma rotina para tratamento de elementos de acordo com seu formato. O algoritmo analyzeRuleConstituent constrói uma tupla ct, composta do seguintes elementos:

- *nrc*: representa o novo constituinte de regra, em conformidade com a sintaxe de geradores de analisadores sintáticos tradicionais;

- *rSet* : representa o conjunto (possivelmente vazio) de novas regras de produções;

- *vSet* : representa o conjunto (possivelmente vazio) de novas variáveis.

O novo constituinte de regra nrc é concatenado à nova regra nr em construção (Linha 4). O conjunto com as novas regras $rSet$, resultantes da análise de rc, é unido ao conjunto de regras da gramática $R'[G']$ em formação, assim como o conjunto com as novas variáveis de gramática $vSet$ é unido ao conjunto de variáveis de $V'[G']$.

O algoritmo analyzeRuleConstituent transforma o constituinte de uma regra para a notação *BNF* criando, se necessário, novas produções e variáveis de gramática.

```
analyzeRuleConstituent(rc)
 1  case rc of
 2      (λ):  nrc[ct] ← λ
 3            rSet[ct] ← ∅
 4            vSet[ct] ← ∅
 5            return (ct)
 6
 7      (empty): nrc[ct] ← λ
 8            rSet[ct] ← ∅
 9            vSet[ct] ← ∅
10            return (ct)
11
12      (v ∈ V[G]): nrc[ct] ← v
13            rSet[ct] ← ∅
14            vSet[ct] ← ∅
15            return (ct)
16
17      (t ∈ T[G])): nrc[ct] ← t
18            rSet[ct] ← ∅
19            vSet[ct] ← ∅
20            return (ct)
21
22      (α*): v ← newVar
23            return analyzeKleeneStar(α)
24
25      (α+): v ← newVar
26            return analyzeKleenePlus(α)
27
28      (α * −t):
29            return analizeTimesSeparator(α)
30
31      (α + −t):
32            return analizePlusSeparator(α)
```

Nos casos em que o constituinte de regra, *rc*, é simplesmente λ, *empty* ou um identificador de *token* ou variável, nada precisa ser feito, e o algoritmo apenas retorna uma tupla com o próprio constituinte e os conjuntos $rSet$ e $vSet$ vazios.

Para os casos em que o constituinte de regra é acompanhado por símbolos que representam repetição, o algoritmo analyzeRuleConstituent executa uma rotina auxiliar. Para os operadores `"*"` , `"+"` , `"*-"` e `"+-"` , os algoritmos analyzeKleeneStar, analyzeKleenePlus, analyzeTimesSeparator e analyzePlusSeparator criam novas variáveis de gramática e novas regras que produzem o efeito esperado desses operadores. A tupla *ct* construída é formada por uma nova variável, pelo conjunto das novas regras de produção $rSet$, e pelo conjunto das novas variáveis

vSet. Note que o constituinte de regra original α é utilizado pelas novas regras de produção criadas, mas α não é usado na regra de produção original, sendo substituído pela variável v criada por meio da atribuição: $nrc[ct] \leftarrow v$. Dessa maneira, as regras de produção contendo símbolos especiais de repetição de *Notus* são transformados em regras *BNF*. A gramática G' resultante do algoritmo incorpora as novas variáveis e regras de produção criadas.

```
analyzeKleeneStar(α)
1  v ← newVar
2  rSet[ct] ← {v → v α, v → λ}
3  vSet[ct] ← {v}
4  nrc[ct] ← v
5  return (ct)
```

```
analyzeKleenePlus(α)
1  v ← newVar
2  rSet[ct] ← {v → v  α, v → α}
3  vSet[ct] ← {v}
4  nrc[ct] ← v
5  return (ct)
```

```
analyzeTimesSeparator(α)
1  v ← newVar
2  v1 ← newVar
3  rSet[ct] ← {v → v1, v → λ, v1 → v1 t α , v1 → α}
4  vSet[ct] ← {v, v1}
5  nrc[ct] ← v
6  return (ct)
```

```
analyzePlusSeparator(α)
1  v ← newVar
2  rSet[ct] ← {v → v t α, v → α}
3  vSet[ct] ← {v}
4  nrc[ct] ← v
5  return (ct)
```

O algoritmo NEWVAR cria um novo identificador para variáveis concatenando, em *var__*, um número inteiro de um contador global do compilador de *Notus*, de modo que se cria um identificador único.

```
NEWVAR()
1  globalCont ← globalCont + 1
2  return "var__"+globalCont
```

Para exemplificar o funcionamento da transformação das regras de produção, considere a gramática $G = (\{E\}, \{x, f, \text{“,”}, \text{“(”}, \text{“)”}\}, \{R\}, E)$ onde,

$$R \ : \ E \ \rightarrow \ x \mid f \ \text{“(”} \ E \ast\!- \ \text{“,”} \ \text{“)”}$$

A conversão dessa gramática via o algoritmo TRANSFORMGRAMMAR produz a gramática:

$$G' \ = \ (\{E, v, v_1\}, \{x, f, \texttt{","}, \texttt{"("}, \texttt{")"}\}, \{R'\}, E),$$

onde

$$\begin{aligned} \texttt{R'} \ : \ & E \ \rightarrow \ x \ \texttt{|} \ f \ \texttt{"("} \ v \ \texttt{")"} \\ & v \ \rightarrow \ v_1 \ \texttt{|} \ \lambda \\ & v_1 \ \rightarrow \ v_1 \ \texttt{","} \ E \ \texttt{|} \ E \end{aligned}$$

4.2.3 Geração da Gramática Abstrata

Após a geração de uma gramática em conformidade com a notação *BNF*, o compilador de *Notus* gera a sintaxe abstrata da linguagem. Dada uma gramática concreta de *Notus* $G = (V, T, R, S)$, o algoritmo GENABSTRACTSYNTAX gera uma gramática abstrata G' de acordo com G. O algoritmo GENABSTRACTSYNTAX recebe o conjunto de regras de produção *pSet* da especificação sintática, e verifica, em cada regra de produção $p \in pSet$, a existência de uma definição de regra abstrata. Uma regra de produção p é uma tupla composta dos seguintes elementos:

- *lhs*: representa o lado esquerdo da produção, ou seja, a variável da gramática;
- *rhs*: representa o lado direito da produção, que é composta por uma seqüência de identificadores de *tokens* ou variáveis, e por *tokens* anônimos da gramática concreta entre aspas;

- *abs*: representa a regra abstrata de uma produção, composta por uma seqüência (possivelmente vazia) de identificadores presentes em *rhs* e por *tokens* anônimos da gramática abstrata, entre aspas.

Se existir uma regra de produção abstrata *abs*, o algoritmo genAbsRule é chamado para cada constituinte abstrato da regra (Linha 6). Caso não exista uma definição de gramática abstrata, o algoritmo genAbsRule é chamado para cada constituinte de regra concreta (Linha 10).

```
genAbstractSyntax(pSet)
 1  createIgnoreGraph(pSet)
 2  for each p ∈ pSet
 3     then if abs[p] ≠ nil ∧ ignoreIndirection(abs[p])
 4             then createEdge(lhs[p],abs[p])
 5             else  if abs[p] ≠ nil
 6                      then for each absc ∈ abs[p]
 7                                do abst ← genAbsRule(absc)
 8                                   nabsR ← nabsR + nac[abst]
 9                                   G' ← updateAbsGrammar(abst,G')
10                      else  for each rc ∈ rhs[p]
11                                do abst ← genAbsRule(rc)
12                                   nabsR ← nabsR + nac[abst]
13                                   G' ← updateAbsGrammar(abst,G')
14                   R[G'] ← nAbsR ∪ R[G']
15  return G'
```

```
updateAbsGrammar(abst, G')
1  R[G'] ← R[G'] arSet[abst]
2  V[G'] ← V[G'] avSet[abst]
3  T[G'] ← T[G']) atSet[abst]
```

O algoritmo genAbsRule retorna uma tupla *abst* composta por:

- *nac*: representa o novo constituinte da regra da gramática abstrata;
- arSet: representa o conjunto (possivelmente vazio) de novas regras de produção da gramática abstrata em formação;
- *avSet*: representa o conjunto (possivelmente vazio) de novas variáveis da gramática abstrata em formação;

```
genAbsRule(c)
 1    (λ ∨ empty): nac[abst] ← empty
 2          arSet[abst] ← ∅
 3          avSet[abst] ← ∅
 4          atSet[abst] ← {empty}
 5          return abst
 6    (v ∈ V[G]): d ← getDomain(v)
 7          nac[abst] ← d
 8          arSet[abst] ← ∅
 9          avSet[abst] ← {d}
10          atSet[abst] ← ∅
11          return abst
12    (t ∈ T[G]): d ← getDomain(t)
13          if d ≠ nil
14            then nac[abst] ← d
15                 arSet[abst] ← {d → t}
16                 avSet[abst] ← {d}
17            else nac[abst] ← t
18                 arSet[abst] ← ∅
19                 avSet[abst] ← ∅
20          atSet[abst] ← {t}
21          return abst
22    (α∗ ∨ α∗−t): abst ← genAbsRule(α)
23          nac[abst] ← nac[abst]∗
24          return abst
25    (α+ ∨ α+−t): abst ← genAbsRule(α)
26          nac[abst] ← nac[abst]+
27          return abst
```

- *atSet*: representa o conjunto (possivelmente vazio) dos novos terminais da gramática abstrata em formação.

O algoritmo genAbsRule é chamado tanto para um constituinte de regra abstrata quanto para constituintes de regra concreta. O constituinte de uma regra abstrata pode ser um identificador de *token* ou de variável seguido, opcionalmente, dos operadores de repetição `"+"` ou `"*"` , ou um *token* anônimo entre aspas duplas. Assim, o algoritmo genAbsRule cria uma tupla *abst*, de acordo com a forma do parâmetro *c*, que representa um constituinte de regra abstrata. No entanto, como o parâmetro *c* do algoritmo genAbsRule pode ser um constituinte de regra de produção concreta, novos casos devem ser considerados: (i) *lambda* e *empty*, nos quais o algoritmo se comporta da mesma maneira, ambos tratados pelo trecho de código entre as Linhas 1 e 5; (ii) identificadores seguidos dos operadores `"*-"` e `"+-"` , para os quais o comportamento esperado é semelhante, respectivamente, aos dos operadores $*$ e $+$. O separador *t*

que acompanha os operadores $*-$ e $+-$ não faz parte da gramática abstrata. O procedimento getDomain, utilizado nesse algoritmo, recupera o domínio de um dado identificador.

Observando-se os casos das Linhas 6 e 12 do algoritmo genAbsRule, que são os que realmente criam o novo constituinte abstrato *nac*, a gramática abstrata é composta pelos domínios dos constituintes das regras, exceto no caso em que um *token* não possui domínio, quando o constituinte abstrato é o próprio *token*.

Para exemplificar a construção da sintaxe abstrata pelo algoritmo genAbsRule, considere a Listagem 4.5, que define uma gramática para uma linguagem de expressões.

```
exp ::= exp "+" term            \\ rule 1
      | exp "-" term            \\ rule 2
      | term : term ;           \\ rule 3
term : Exp ::= term "*" factor  \\ rule 4
             | term "/" factor  \\ rule 5
             | factor : factor  \\ rule 6
factor : Exp ::= "(" exp ")"    \\ rule 7
               | id             \\ rule 8
               | num ;          \\ rule 9
```

Listagem 4.5: Especificação de uma gramática expressões em *Notus*

A gramática abstrata para essa linguagem é $G' = (V', T', R', Exp)$, onde:

- $V' = \{\texttt{Id}, \texttt{Num}, \texttt{Exp}\}$
- $T' = \{\texttt{"+"}, \texttt{"-"}, \texttt{"*"}, \texttt{"/"}, \texttt{"("}, \texttt{")"}, \texttt{id}, \texttt{num}\}$
- R' :

```
Exp → Exp "+" Exp
    | Exp "-" Exp
    | Exp "*" Exp
    | Exp "/" Exp
    | "(" Exp ")"
    | Id
    | Num
Id  → id
Num → num
```

Para exemplificar a geração da gramática abstrata quando uma especificação alternativa é desejada, considere o código da Listagem 4.4, p.80.

A gramática abstrata para essa linguagem é $G' = (V', T', R', Exp)$, onde:

- $V' = \{\texttt{Id}, \texttt{Num}, \texttt{Exp}\}$

- $T' = \{$`"add"`, `"sub"`, `"mul"`, `"div"`, `"("`, `")"`, `id`, `num`$\}$

- R' :
```
Exp  →  "add" Exp Exp
     |  "sub" Exp Exp
     |  "mul" Exp Exp
     |  "div" Exp Exp
     |  id
     |  num
Id   →  id
Num  →  num
```

Note que as regras 3, 6, 7, 8 e 9 definem regras que ignoram a indireção. Para a regra 7 da gramática concreta, a regra `Exp` → `id` é incluída na gramática abstrata, já que existe a regra `Id` → `id`. De forma análoga, para a regra 8, a regra `Exp` → `num` é incluída na gramática abstrata. As regras 3, 6 e 9 não geram novos componentes na gramática abstrata, pois os elementos dessas regras, `term`, `factor` e `exp` respectivamente, possuem o mesmo domínio da variável da gramática concreta.

A Figura 4.13 mostra os efeitos das declarações das regras 7 e 8, que ignoram a indireção para a gramática da Listagem 4.4, em comparação à gramática da Listagem 4.5. As linhas tracejadas correspondem às declarações das regras para ignorar indireção. O efeito da regra de ignorar indireção é criar uma ligação direta entre o domínio da variável da gramática `Exp` e os elementos de indireção `id` na regra 7, e `num`, na 8. As linhas contínuas mostram a ligação entre os elementos da gramática abstrata para as regras 7 e 8 da gramática da Listagem 4.5.

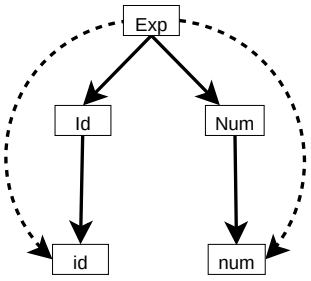

Figura 4.13: Representação gráfica das gramáticas abstratas geradas para as regras 7 e 8 das Listagens 4.4 e 4.5

A inclusão das regras abstratas, devida à definição de regras para ignorar indireção, é modelada por meio de um grafo $G_I = (V, E)$, onde V é o conjunto formado pela união de todas as variáveis e terminais da gramática concreta de uma linguagem, e uma aresta $e = (v_1, v_2) \in E$ é criada para cada produção na forma: $v_1 \rightarrow \alpha : v_2$, onde os domínios de v_1 e v_2 são distintos. Esse grafo é construído durante o

processo de geração da gramática abstrata pelo algoritmo genAbstractSyntax, p.87, onde a Linha 1 cria um grafo G_I sem arestas, e a Linha 4 adiciona uma aresta ao grafo entre o lado esquerdo da produção `lhs` e a regra abstrata `abs`, já que uma regra abstrata para ignorar indireção é composta por um único elemento. O algoritmo genAbstractSyntax cria as regras da gramática abstrata desconsiderando as regras para ignorar indireção e o grafo que representa a relação entre as variáveis e *tokens* da gramática concreta para regras desse tipo.

Para exemplificar a presença de vários níveis de indireção, considere a gramática $G = (V, T, R, p)$, onde:

- $V = \{\mathtt{p}, \mathtt{q}, \mathtt{r}, \mathtt{x}, \mathtt{y}\}$
- $T = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$
- R definida pela Listagem 4.6.

```
p ::= q           \\ rule 1
    | r : r       \\ rule 2
    | a : a ;     \\ rule 3
q ::= z           \\ rule 4
r ::= b c         \\ rule 5
    | x : x       \\ rule 6
x ::= y : y       \\ rule 7
y ::= w           \\ rule 8
```

Listagem 4.6: Especificação de uma gramática com vários níveis de indireção

Assim, o algoritmo genAbstractSyntax gera:

- o grafo correspondente a essa gramática (Figura 4.14);
- as regras de produção da gramática abstrata, desconsiderando as produções geradas pelas regras de indireção dada a seguir. Os domínios de *tokens* e variáveis são obtidos capitalizando-se a primeira letra destes elementos da gramática; assim, `P` é o domínio da variável `p`.

$$
\begin{aligned}
P &\rightarrow Q \\
Q &\rightarrow Z \\
R &\rightarrow B\ C \\
X &\rightarrow \\
Y &\rightarrow W \\
A &\rightarrow a
\end{aligned}
$$

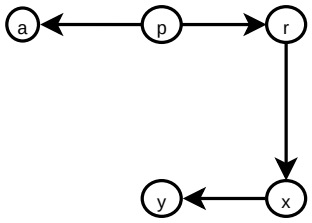

Figura 4.14: Grafo G_I gerado para gramática da Listagem 4.6

Após a obtenção do grafo e de parte da gramática abstrata, calcula-se o fecho transitivo direto do grafo gerado G_I, $\hat{\Gamma}^+(G_I)$, obtendo-se um novo grafo G'_I. A Figura 4.15 mostra o grafo $G'_I = \hat{\Gamma}^+(G_I)$, onde G_I é o grafo da Figura 4.14.

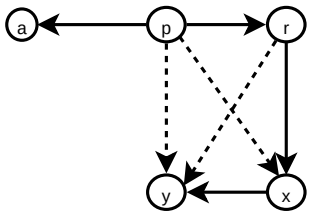

Figura 4.15: Grafo G'_I correspondente ao fecho transitivo direto de G_I

As arestas do grafo G'_I representam as conexões entre variáveis da gramática concreta, em que as arestas pontilhadas são decorrentes do cálculo do fecho transitivo direto. De acordo com esse grafo a variável `p` deve incorporar as produções de `a`, `r`, `x` e `y`. A variável `r` deve incorporar as produções das variáveis `x` e `y`, e a variável `x` deve incorporar as produções da variável `y`. Ao se completar a gramática abstrata com as novas produções devidas às definições de regra de indireção, obtém-se a gramática abstrata final $G' = (V', T', R', P)$, onde:

- $V' = \{P, Q, R, X, Y, A, B, C\}$
- $T' = \{a, b, c\}$
- R':

$$\begin{array}{rcl} P & \rightarrow & Q \\ & | & B\ C \\ & | & D_w \\ & | & a \\ Q & \rightarrow & Z \\ R & \rightarrow & B\ C \\ & | & W \\ X & \rightarrow & W \\ Y & \rightarrow & W \\ A & \rightarrow & a \end{array}$$

4.2.4 Simplificação da Gramática

O compilador de Notus simplifica a gramática concreta $G = (V, T, R, S)$ de uma especificação, gerando uma versão equivalente de G, usando algoritmos para eliminação de símbolos inúteis adaptados de [Sipser, 1996].

Algoritmo I: Eliminação de Variáveis que Não Geram Strings. O algoritmo para eliminação de variáveis que não geram *strings* cria, a partir da gramática concreta $G = (V, T, R, S)$ produzida pela etapa de **Grammar analyzer** da Figura 4.12, uma nova gramática $G = (V', T, R', S)$, onde V' contém somente as variáveis que geram *string*, e R' é o conjunto das regras de produção contendo somente *tokens* e variáveis $v \in V'$.

Para que as variáveis que não geram *strings* sejam eliminadas são construídos dois conjuntos globais denominados *variableLists* e *generateStringsVar*. O conjunto *generateStringsVar* armazena identificadores de variáveis que geram *strings*, e o conjunto *variableLists* contém tuplas *vlt* formadas por uma variável v e um conjunto $pSet$ de conjuntos $vSet$, onde $vSet$ é o conjunto de variáveis de uma produção $p \in pSet$. Por exemplo, a variável *exp*, da definição da Figura 4.5, será representa pela tupla $vtl = (exp, \{\{exp, term\}, \{exp, term\}, \{factor\}\})$.

O algoritmo createMapVarToSetOfVar cria os conjuntos *variableLists* e *generateStringsVar*. Inicialmente, esse algoritmo constrói, para cada variável recebida como parâmetro, uma tupla *vtl*. Ao se percorrer, na Linha 3, cada regra de produção $p \in ps$ da variável v, é criado o conjunto de variáveis $vSet$ referente à produção p analisada. O teste da Linha 7 garante que os constituintes da regra concreta rc a serem inseridos em $vSet$ são identificadores de variáveis. O *flag isTokenRule* indica se a produção analisada gera *string*. Isso ocorre quando uma produção contém somente *tokens*, assim a variável v que possui pelo menos um conjunto $vSet$ vazio gera *string*. Dessa maneira, uma variável v nessa situação é adicionada no conjunto *genStrings* na Linha 13.

Após a criação dos conjuntos globais *variableLists* e *generateStringsVar*, o algoritmo eliminateVariables é executado, percorrendo cada tupla $vlt \in variableList$, e para cada conjunto $vSet \in pSet$ da tupla analisada removem-se os identificadores de variáveis que geram *strings*. O teste da Linha 5 verifica se a diferença dos conjuntos $vSet$ e *genStrings* é vazia, indicando a existência de uma nova variável que gera *string*, pois, nesse caso, o conjunto $vSet$ é composto por variáveis que geram *strings* e, portanto, a variável v, da tupla de $vSet$, gera *string*. Logo, essa variável v é incluída na lista *generateString*, na Linha 6, e o *flag modify* é sinalizado positivamente indicando a necessidade da repetição do processo descrito.

```
createMapVarToSetOfVar(v)
 1  v[vlt] ← v
 2  ps ← getProductions(v)
 3  for each p ∈ ps
 4       do vSet[vlt] ← ∅
 5          isTokenRule ← true
 6          for each rc ∈ rhs[p]
 7               do if isVariable(rc)
 8                    then vSet[vlt] ← vSet[vlt] ∪ rc
 9                         isTokenRule ← false
10          pSet[vlt] ← pSet[vlt] ∪ vSet[vlt]
11  variableLists ← variableLists ∪ vlt
12  if isTokenRule
13     then genStrings ← genStrings ∪ v
```

```
eliminateVariables()
1  repeat  modify ← false
2          for each vlt ∈ variableLists
3               do for each vSet ∈ pSet[vlt]
4                    do vSet ← vSet\genStrings
5                       if vSet = ∅
6                         then generateString ← generateString ∪ v[vlt]
7                              modify ← true
8    until not modify
```

Por exemplo, considere a gramática $G = V, T, R, S$, onde $V = \{A, B, C, D, E, F, S\}$, $T = \{a, b, c, d, e, f, g\}$ e R é:

$$
\begin{aligned}
R: \quad S &\rightarrow AbB \mid Cd \mid Df \\
A &\rightarrow Ac \mid d \\
B &\rightarrow Cd \mid Bc \mid A \\
C &\rightarrow De \mid Cf \\
D &\rightarrow Dg \mid Ee \\
E &\rightarrow Ce \mid Db \mid Ed \\
F &\rightarrow Cb \mid Fab \mid c
\end{aligned}
$$

A Figura 4.16 mostra os conjuntos *variableLists* e *generateStringsVar*, onde os conjuntos *variableLists* 0 e *genStrings* 0 foram criados pelo algoritmo createMapVarToSetOfVar e os conjuntos *variableLists* 1 e *variableLists* 2 são resultantes de cada repetição do algoritmo eliminateVariables.

As variáveis que geram *strings* são A, B, F e S. Assim, a gramática obtida após a simplificação é $G' = (\{A, B, F, S\}, T, R', S)$, onde:

variableLists 0	*variableLists* 1	*variableLists* 2
$(S, \{\{A, B\}, \{C\}, \{D\}\})$	$(S, \{\{B\}, \{C\}, \{D\}\})$	$(S, \{\emptyset, \{C\}, \{D\}\})$
$(A, \{\{A\}, \emptyset\})$	$(A, \{\emptyset\})$	$(A, \{\emptyset\})$
$(B, \{\{C\}, \{B\}, \{A\}\})$	$(B, \{\{C\}, \{B\}, \emptyset\})$	$(B, \{\{C\}, \emptyset\})$
$(C, \{\{D\}, \{C\}\})$	$(C, \{\{D\}, \{C\}\})$	$(C, \{\{D\}, \{C\}\})$
$(D, \{\{D\}, \{E\}\})$	$(D, \{\{D\}, \{E\}\})$	$(D, \{\{D\}, \{E\}\})$
$(E, \{\{C\}, \{D\}, \{E\}\})$	$(E, \{\{C\}, \{D\}, \{E\}\})$	$(E, \{\{C\}, \{D\}, \{E\}\})$
$(F, \{\{C\}, \{F\}, \emptyset\})$	$(F, \{\{C\}, \emptyset\})$	$(F, \{\{C\}, \emptyset\})$
$genStrings\ 0 = \{A, F\}$	$genStrings\ 1 = \{A, B, F\}$	$genStrings\ 2 = \{A, B, F, S\}$

Figura 4.16: Conjuntos *variableLists* e *generateStringsVar* gerado pelos algoritmos createMapVarToSetOfVar e eliminateVariables.

$$
\begin{aligned}
R' : \quad S &\rightarrow AbB \\
A &\rightarrow Ac \mid d \\
B &\rightarrow Bc \mid A \\
F &\rightarrow Fab \mid c
\end{aligned}
$$

Algoritmo 2: Eliminação de Símbolos Não-Alcançáveis. O segundo algoritmo consiste em eliminar terminais e variáveis não alcançáveis a partir do símbolo inicial da gramática. Esse problema é modelado com um grafo dirigido $D = (N, E)$, construído a partir da gramática $G' = (V', T, R', S)$ gerada pelo algoritmo anterior, onde $N = V' \cup T$ e, para cada $n_1, n_2 \in N$, $(n_1, n_2) \in E$, se $n_1 \in V'$ e existe pelo menos uma regra $n_1 \rightarrow \alpha n_2 \beta \in R'$, para algum $\alpha, \beta \in (V' \cup T)^*$. A partir do grafo D gerado pelo Algoritmo 2 é possível gerar uma nova gramática $G'' = (V'', T'', R'', S)$, onde:

- $V'' = V' \cap \hat{\Gamma}^+(S)$, onde $\hat{\Gamma}^+(S)$ é o fecho transitivo direto do vértice S no grafo H;
- $T'' = T \cap \hat{\Gamma}^+(S)$; representa todos os terminais alcançáveis a partir de S;
- $R'' = \{A \rightarrow \alpha \in R' \mid A \in V'' \wedge \alpha \in (V'' \cup T'')^* \wedge \alpha \neq A\}$; representa todas as regras de produção úteis de G.

Assim, a gramática G'' é obtida calculando-se o fecho transitivo do vértice que representa o símbolo inicial da gramática S. O algoritmo utilizado para o cálculo de $\hat{\Gamma}^+(S)$ foi o algoritmo de *Floyd Warshall* [Sedgewick, 1990, Capítulo 32].

Para a gramática $G' = (\{A, B, F, S\}, T, R', S)$ produzida pelo exemplo do Algoritmo 1, o Algoritmo 2 cria o grafo da Figura 4.17

Para o grafo da Figura 4.17, o fecho transitivo direto de S é $\hat{\Gamma}^+(S) = \{A, B, S, b, c, d\}$, e a gramática gerada pelo Algoritmo 2 é $G'' = (\{A, B, S\}, \{b, c, d\}, R'', S)$, onde:

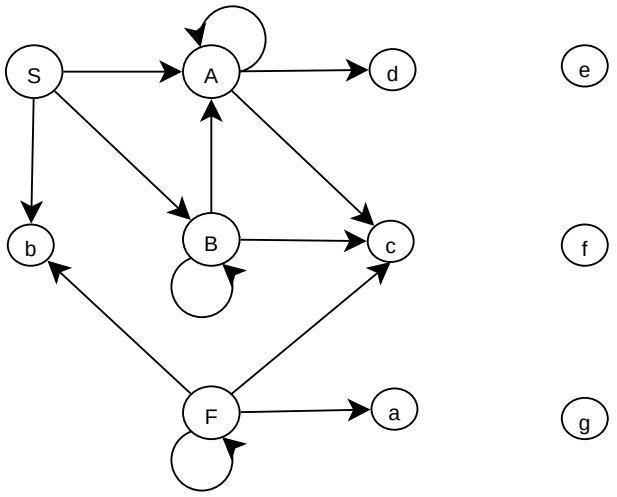

Figura 4.17: Grafo gerado pelo Algoritmo 2 para a gramática G' gerada pelo Algoritmo 1

$$
\begin{array}{lrcl}
R'': & S & \rightarrow & AbB \\
 & A & \rightarrow & Ac \mid d \\
 & B & \rightarrow & Bc \mid A
\end{array}
$$

4.2.5 Domínios Sintáticos

Cada regra da gramática abstrata estende os domínios de sua variável do lado esquerdo, incluindo na sua definição um novo termo, derivado de seu lado direito. Por exemplo, na gramática da Listagem 4.4, p.80, o domínio *Exp* das variáveis *exp*, *term* e *factor* é estendido, devido às regras da gramática abstrata, para:

$$
\begin{array}{rcl}
\texttt{Exp} & = & \texttt{Id} + \texttt{Num} \\
 & & + (\{\texttt{"add"}\} \times \texttt{Exp} \times \texttt{Exp}) \\
 & & + (\{\texttt{"sub"}\} \times \texttt{Exp} \times \texttt{Exp}) \\
 & & + (\{\texttt{"mul"}\} \times \texttt{Exp} \times \texttt{Exp}) \\
 & & + (\{\texttt{"div"}\} \times \texttt{Exp} \times \texttt{Exp})
\end{array}
$$

Os domínios formados pelas regras da gramática abstrata são representados no arquivo *Sym.y* por *datatypes* em *Haskell*. Assim, para cada variável sintática presente na especificação da linguagem, é criado um *datatype* e, para cada regra de produção da gramática abstrata dessa variável, é criado um construtor de tipos contendo valores para cada constituinte da regra. A construção de *datatypes* é realizada em dois passos, criação de *datatypes* para *tokens* e criação de *datatypes* para variáveis.

Criação de *datatypes* para *tokens*. Durante a geração do arquivo *Lex.x* (veja Seção 4.1.4) cria-se um *datatype* que representa todos os *tokens* presentes na descrição. Adicionalmente, devem-se criar *datatypes* para os *tokens* que possuem domínios definidos pelo projetista. Um *datatype* é uma tupla *dt* composta pelo nome do *datatype* *ndt* e por um conjunto de valores *dvSet*. O algoritmo createTokenDataType é responsável pela criação dos *datatypes* para os *tokens* com domínios definidos. Inicialmente, o algoritmo recupera o domínio *tDom* de cada *token* da definição, e caso esse não seja nulo, o domínio do *token* nomeia o *datatype dt* em formação na Linha 4. A seguir, o algoritmo recupera a função de lexema associada ao *token* t, e chama o procedimento createDataValue passando a função de lexema *tFun* e o *datatype dt* em construção. O procedimento createDataValue testa se a função fun é nula e, caso isso ocorra, um valor de *datatype* composto, por um novo construtor e pelo tipo *String*, é adicionado ao conjunto de valores de *datatypes* *dvSet*, na Linha 2. Mas, se existir a definição de uma função de lexema, então o valor do *datatype* é construído concatenando-se um novo construtor de tipos ao tipo de retorno da função definida na Linha 3.

```
createTokenDataType(G)
 1  for each t ∈ T[G]
 2       do tDom ← getDomain(t)
 3          if tDom ≠ nil
 4             then ndt[dt] ← tDom
 5                  tFun ← getTokenFunction(t)
 6                  dt ← createDataValue(tFun,dt)
 7                  tExtensions ← getTokenExtensions
 8                  for each extension ∈ tokenExtensions
 9                       do extensionFun ← getFunction(extension)
10                          dt ← createDataValue(extensionFun,dt)
```

```
createDataValue(dt, fun)
1  if fun = nil
2     then dvSet[dt] ← dvSet[dt] ∪ newDataConstructor(dt) + "String"
3     else dvSet[dt] ← dvSet[dt] ∪ newDataConstructor(dt) +
4          getFunctionReturnDomain(tFun) return dt
```

A Linha 7 do procedimento createTokenDataType recupera as possíveis extensões de *tokens* definidas para o *token* t em questão. Cada extensão de *token* é analisada e, de acordo com a existência ou não de uma função de lexema definida, seus valores de *datatype* são construídos.

Para exemplificar, considere os *tokens num* e *id*, além das funções de lexema *getNumber*, *getId* e *getNumberInHexa* mostrados na Listagem 4.7.

```
1 // getNumber :: String → Int
2 // getId :: String → String
3 // getNumberInHexa :: String → Int
4
5 token num : Num = [0−9]+ is getNumber;
6 token id : Id = [a−z]+[0−9]* is getId ;
7
8 extend token num with "0x"[0−9A−Fa−f]+ is getNumberInHexa ;
9 extend token id with "_"[a−z]+[0−9]* ;
```

Listagem 4.7: Exemplo de definição de *tokens*: *id* e *num*

Os *datatypes* criados de acordo com o algoritmo CREATETOKENDATATYPE são:

```
data Id  = Id__7 String
         | Id__8 String
data Num = Num__9 Int
         | Num__10 Int
```

Criação de *datatypes* para variáveis: A criação de *datatypes* para variáveis é realizada sobre cada regra de produção da gramática abstrata pelo algoritmo CREATEVARDATATYPES. Inicialmente, o algoritmo recupera as produções *ps* de cada variável *v*. A seguir, na Linha 6, este algoritmo verifica se a regra abstrata *abs* é um identificador de *token*, isso ocorre quando há a definição de uma regra para ignorar indireção (Veja Exemplos da Seção 4.2.3). Nesse caso, o *datatype* a ser gerado deve incorporar os valores de *datatypes* desse *token* (Linhas 6 a 8). Outra forma de uma regra abstrata é um *string* contendo identificadores de domínios e *tokens* anônimos. Nesse caso, o *datatype* gerado é constituído pelos identificadores de domínios ignorando-se os *tokens* anônimos, pois estes não contêm valores a serem armazenados (Linhas 10 a 16). Caso o identificador de domínio seja seguido dos operadores que indicam repetição + ou ∗, uma lista desse domínio é criada pela inserção dos símbolos "[" e "]", já que esses operadores indicam várias ocorrências do identificador que o precede.

Para exemplificar a geração de *datatypes* para variáveis, considere a gramática abstrata da Página 89 referente à gramática concreta da Listagem 4.4. O *datatype* gerado para `Exp` é mostrado na Figura 4.18.

Os valores do *datatype Exp* `Exp19 String` e `Exp20 String`, presentes no *datatype* `Exp`, são relativos à presença do *token* `id` na gramática abstrata, e os valores `Exp21 Int` e `Exp22 Int` devidos ao *token* `num`.

```
createVarDataTypes(G)
 1  for each v ∈ V[G])
 2       do vDom ← getDomain(v)
 3          ndt[dt] ← vDom
 4          ps ← getProductions(v)
 5          for each p ∈ ps
 6               do if isTokenIdentifier(abs[p])
 7                     then tdv ← getDataValue(abs[p])
 8                          dvSet[dt] ← dvSet[dt] ∪ newDataConstructor(dt) + tdv
 9                     else newdv ← newDataConstructor(dt)
10                          for each absc ∈ (abs[p])
11                               do case absc of
12                                    (α): if isDomain(α)
13                                      then newdv ← newdv + absc
14                                    (α+ ∨ α∗): if isDomain(α)
15                                      then newdv ← newdv + "[" absc"]"
16                          dvSet[dt] ← dvSet[dt] ∪ newdv
```

```
exp ::= exp"+"term :["add" exp term]
      | exp"-"term :["sub" exp term]
      | term : term ;
term ::= term"*"factor
                 :["mul" term factor]
       | term "/" factor
                 :["div" term factor]
       | factor : factor ;
factor ::=  id : id
         | num : num
         | "(" exp ")" : exp;
```

```
data Exp = Exp11 Exp Exp
         | Exp12 Exp Exp
         | Exp13 Exp Exp
         | Exp14 Exp Exp
         | Exp15 String
         | Exp16 String
         | Exp17 Int
         | Exp18 Int
```

Figura 4.18: Exemplo de criação de *datatypes* para variáveis.

4.2.6 Geração de Ações Semânticas

Após a geração dos *datatypes*, o compilador de *Notus* possui todas as informações necessárias para a criação das ações semânticas da gramática a ser gerada. Assim o compilador associa a cada produção da gramática concreta uma ação a ser realizada quando o analisador sintático casar a seqüência de símbolos presentes no lado direito dessa regra de produção. As ações semânticas geradas pelo compilador de *Notus* criam nodos da árvore de sintaxe abstrata, que são valores dos *datatypes* definidos na etapa **Datatypes generation** da Figura 4.12. A forma de uma regra de produção para o

Happy é:

$$\texttt{n} \ : \ \texttt{t}_1 \ \ldots \ \texttt{t}_n \quad \{ \ \texttt{A} \ \}$$

onde `n` é o nome de uma variável, $\texttt{t}_1 \ \ldots \ \texttt{t}_n$ é uma seqüência de símbolos terminais e não-terminais da linguagem, e `A` é a ação semântica, que pode referenciar os valores de cada $\texttt{t}_i$ via `$i`.

O algoritmo createVarSemanticActions cria a ação semântica correspondente a cada produção, construindo um par *pp*, composto pela regra de produção concreta, *rhs*, e pela ação semântica *sa*. O algoritmo recupera as produções *ps* de cada variável e, para cada produção $p \in ps$, o algoritmo testa se a regra abstrata *abs* é uma regra de ignorar indireção (Linha 5). Caso isso ocorra, a ação semântica formada consiste em chamar uma função que fará o mapeamento do domínio do identificador da indireção para o domínio da variável. O parâmetro para a função *ignoreIndirectionFun* é obtido consultando-se, na regra concreta, qual é a posição do identificador do constituinte abstrato. A ação semântica é então formada pela concatenação do nome da função, retornado pelo método getNewignoreIndirectionFun, com a posição do constituinte abstrato (Linha 7).

Caso a regra abstrata não seja uma regra para ignorar indireção, cada constituinte da regra abstrata é acessado e sua posição é consultada na regra concreta (Linha 11). O construtor do *datatype* referente à regra de produção analisada concatenado com cada posição do constituinte abstrato forma a ação semântica de cada produção (Linhas 9 e 12). O par *pp* é construído na Linha 14.

```
createVarSemanticActions(G)
 1  for each v ∈ V[G]
 2       do ps ← getProductions(v)
 3          for i ← 0 to length[ps]
 4               do p ← getProduction(v,i)
 5                  if isIgnoreIndirection(abs[p])
 6                       do pos ← getPosition(abs[p],rhs[p])
 7                          semAction ← getNewignoreIndirectionFun + ("$"+pos)
 8                    else  dv ← getDataValue(v,i)
 9                          semAction ← getDataValueConstructor(dv)
10                          for each absc ∈ abs[p]
11                               do pos ← getPosition(absc,rhs[p])
12                                  semAction ← semAction + ("$"+pos)
13                  rhs[sap] ← rhs[p]
14                  sa[sap] ← semAction
```

Para exemplificar a geração das ações semânticas, considere a gramática da Lista-

gem 4.4, p.80. As produções geradas para essa gramática são:

```
exp    : exp "+" term     {Exp11 $1 $3}
       | exp "-" term     {Exp12 $1 $3}
       | term             {Exp13 $1}
term   : term "*" factor  {Exp14 $1 $3}
       | term "/" factor  {Exp15 $1 $3}
       | factor           {Exp16 $1}
factor : "(" exp ")"      {Exp17 $2}
       | id               {ignoreIndirectionFun25 $1}
       | num              {ignoreIndirectionFun26 $1}
```

As funções `ignoreIndirectionFun25` e `ignoreIndirectionFun26`, geradas na Linha 7 do algoritmo createVarSemanticActions, são resultantes da definição de uma regra para ignorar indireção na descrição da gramática em *Notus* e, nesse caso, é preciso que se mapeie os elementos do *datatype* do identificador da indireção, que pode identificar um *token* ou uma variável, nos elementos do *datatype* da variável. A função `ignoreIndirectionFun25` mapeia `Id` em `Exp` e `ignoreIndirectionFun26` mapeia `Num` em `Exp`. O código para essas duas funções é mostrado a seguir:

```
ignoreIndirectionFun__25 :: Id -> Exp
ignoreIndirectionFun__25 igInd=
        case igInd of
                (Id__7 a1) -> (Exp__19 a1)
                (Id__8 a1) -> (Exp__20 a1)

ignoreIndirectionFun__26 :: Num -> Exp
ignoreIndirectionFun__26 igInd=
        case igInd of
                (Num__9 a1) -> (Exp__21 a1)
                (Num__10 a1) -> (Exp__22 a1)
```

4.2.7 Geração de Código

A Figura 4.19 descreve, em semântica denotacional, as equações para a geração de uma gramática para o gerador de analisadores sintáticos *Happy* com base nas informações geradas pela análise semântica, descrita na Seção 4.2.1, de uma especificação sintática em *Notus*.

Domínio Sintático:

PP = $\{(RHS, SA) \mid (RHS, SA)$ é o par criado pelo algoritmo createVarSemanticActions onde, RHS é a regra de produção concreta, e SA é a ação semântica.$\}$

Sintaxe Abstrata:

$$\begin{array}{lll} D & ::= & \ldots \\ & \mid & I_1 \ : \ I_2? \ \text{“ ::= ”} \ PP \\ PP & ::= & (S_1, S_2) \ \text{“|”} \ (RHS, SA) \mid (S_1, S_2) \end{array}$$

Funções Semânticas de Compilação:

- Compilação de Regra
  $\mathcal{KPP} : \mathbf{PP} \mapsto \mathbf{IO}$

Equações Semânticas Para Geração de Código:

$$\begin{array}{lcl} \mathcal{KD}[\![I_1 : I_2? = R \ ]\!]s_0 & = & print(I_1) \bullet print(\text{“ : ”}) \bullet \mathcal{KR}[\![R]\!] \\ \\ \mathcal{KPP}[\![(S_1, S_2) \text{“|”} PP]\!] & = & print(S_1) \bullet print(\text{“\{”} S_2 \text{“\}”}) \bullet \\ & & print(\text{“ \textbackslash n | ”}) \bullet \mathcal{KPP}[\![PP]\!] \\ \\ \mathcal{KPP}[\![S_1, S_2]\!] & = & print(S_1) \bullet print(\text{“\{”} S_2 \text{“\}”}) \end{array}$$

Figura 4.19: Regras de geração de código da especificação da gramática em *Notus* para *Happy*.

4.3 Conclusões

Neste capítulo foram apresentadas as técnicas utilizadas na construção do compilador *Notus* para a compilação das especificações léxica e sintática modulares de uma linguagem. Em especial, foram detalhadas as técnicas utilizadas no compilador para a construção do reconhecedor sintático, gerado a partir de definições modulares da sintaxe de um linguagem em *Notus*.

Dificuldades na compilação de especificações modulares surgem a partir da união dos módulos para geração de código, como, por exemplo, determinar a ordem de geração de expressões regulares e macros no analisador léxico. Como essas questões são transparentes para programadores da linguagem *Notus*, elas devem ser resolvidas pelo compilador antes da geração de um reconhecedor sintático para a especificação.

As especificações léxica e sintática são agrupadas para a formação de analisadores léxico e sintático únicos. No processamento da especificação léxica, o compilador *Notus* deve garantir que toda macro é definida antes de seu uso, e que as expressões regulares

que definem os *tokens* da linguagem sejam ordenadas, de forma que expressões mais gerais não sobreponham as menos gerais. Na especificação sintática, o compilador reúne as produções estendidas de uma variável, de forma que não existam diferenças semânticas entre essas e as definidas no momento da declaração da variável.

O capítulo seguinte apresenta a compilação da especificação semântica de uma linguagem para a formação do módulo *Haskell* responsável pela interpretação semântica das construções da linguagem especificada.

Capítulo 5

Compilação da Especificação Semântica

O compilador de *Notus* gera, a partir da definição semântica de uma linguagem L, o módulo *Haskell* principal (*Main.hs*) do interpretador para L. Esse módulo é composto pela definição dos *datatypes* que representam os domínios semânticos da especificação, e a definição das funções que representam a semântica de L. Esta seção descreve o processo que abrange a compilação da especificação semântica de uma linguagem, as verificações realizadas sobre as equações semânticas, e a geração do módulo principal do interpretador para essa linguagem.

5.1 Visão Geral

A Figura 5.1 ilustra os dados produzidos durante a compilação dos componentes semânticos de uma especificação e as etapas da tradução dessa especificação em Notus para um programa em *Haskell*. A definição semântica da linguagem é composta pela definição dos domínios semânticos e pelas equações semânticas que descrevem o significado das construções da linguagem especificada.

Inicialmente, o compilador de *Notus* processa os domínios semânticos, na etapa **Semantic domain analyzer**, gerando *datatypes* em *Haskell* e um grafo de domínios (*domain graph*). O grafo de domínios $G_d = (V_d, E_d)$ representa a relação entre os domínios definidos, onde cada vértice $v \in V_d$ representa um domínio, e cada aresta $e = (u, v) \in E_d$, $(u, v \in V_d)$ indica que o domínio v compõe o domínio u, ou seja, o domínio v é um dos componentes da expressão de domínio de u. Um novo grafo de domínio G'_d é criado calculando-se o fecho transitivo direto do grafo G_d. O grafo G'_d é posteriormente usado na etapa de verificações de tipos das expressões de cada definição de função semântica presente na especificação. A Seção 5.2 detalha o processo

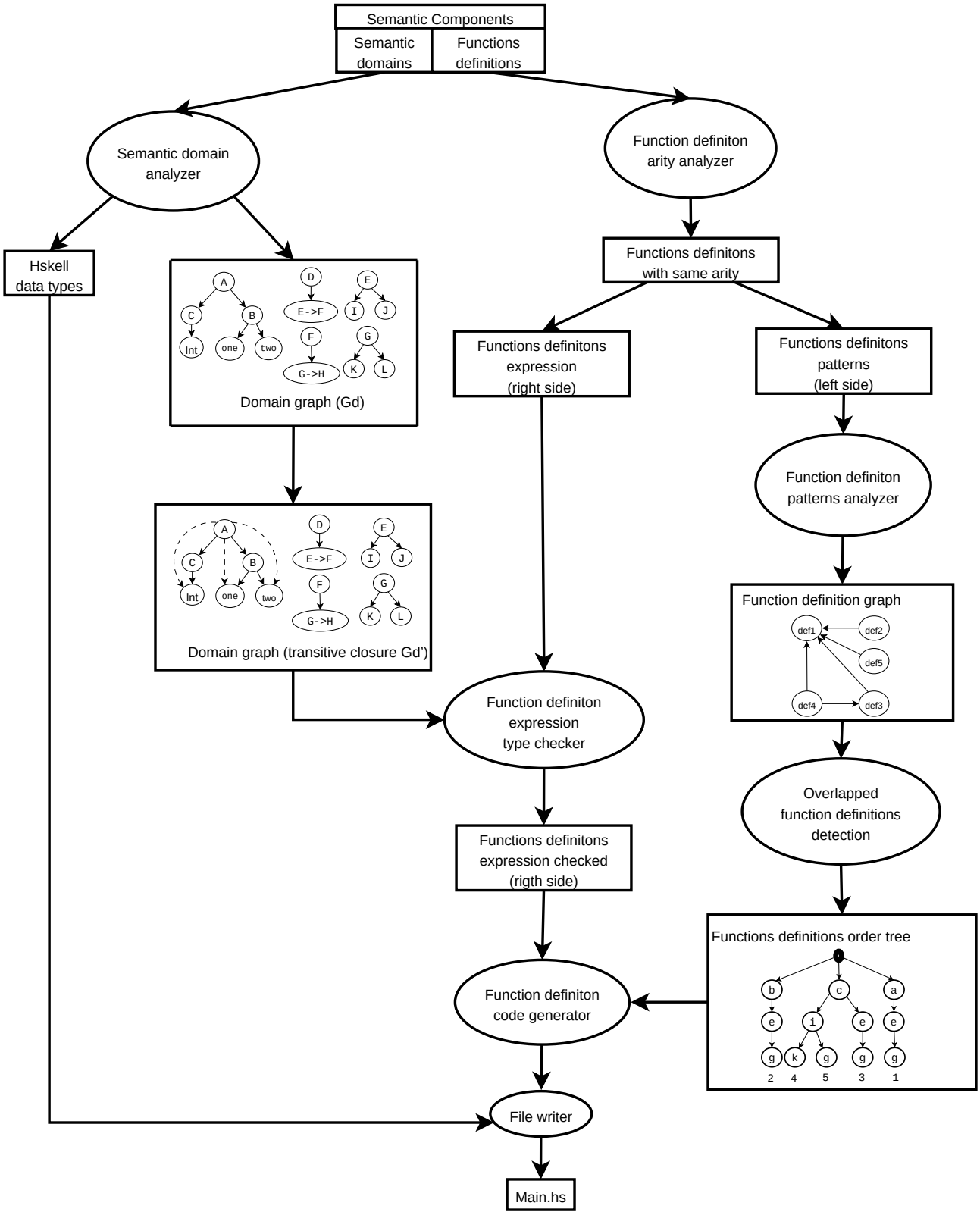

Figura 5.1: Fluxo de dados e etapas de tradução da descrição semântica de uma linguagem em um programa em *Haskell*.

de criação dos *datatypes* e do grafo de domínios.

Notus permite, por motivos de legibilidade e facilidade de escrita, que equações de definições de uma função f omitam parâmetros. Por exemplo, para uma função `f` de assinatura $d_1 \rightarrow d_2 \ \cdots \ \rightarrow d_n$, uma equação de definição de `f` pode ter a forma `f` $p_1\ p_2\ \cdots\ p_k$, onde $k \leq n$ e p_i é um padrão para um argumento de tipo d_i, para $1 \leq i \leq k$. Assim, as definições de funções com parâmetros omitidos são normalizadas pela etapa **Function definitions arity analyzer** (veja Seção 5.4) em definições com o mesmo número de parâmetros, para que essas possam ser processadas pelas etapas seguintes.

As definições de função com mesmo número de parâmetros provenientes da etapa **Function definitions arity analyzer**, têm o lado esquerdo, os parâmetros, e o lado direito, a expressão, analisados separadamente. As expressões das definições passam pela etapa **Function definition expression type checker** (veja Seção 5.5), onde seu tipo é verificado de acordo com o grafo de domínios G_d. Os padrões das definições são analisados pela etapa **Function definition patterns analyzer** (veja Seção 5.6). Essa fase constrói um grafo $G_{\text{def}} = (V_{\text{def}}, E_{\text{def}})$ que representa a relação entre as definições de uma função, onde cada vértice $v \in V_{\text{def}}$ é a seqüência de padrões de uma definição, e cada aresta $e = (u, v) \in E_{\text{def}}$, onde $u, v \in V_{\text{def}}$, indica que a sequência de padrões u deve ser gerada antes da seqüência v.

O grafo G_{def} é usado na etapa **Overlapped function definitions analyzer** (veja Seção 5.7), para identificação de definições sobrepostas. Após a construção do grafo G_{def}, essa etapa constrói uma árvore, que determina a ordem em que as definições de uma função são geradas, de forma a evitar que essas definições se sobreponham no código *Haskell* gerado. A construção dessa árvore é necessária para que as definições de funções escritas em *Notus*, de forma modular, possuam a mesma semântica após a sua tradução para definições em *Haskell*, onde definições são processadas de acordo com sua ordem no código-fonte.

A etapa **Function definition code generator** gera as definições de funções em *Haskell*, utilizando a árvore de ordenação de definições e as expressões já verificadas. As definições são geradas utilizando uma expressão **case**, e que cada nível na árvore de definições é traduzido para uma *abstração lambda* em *Haskell*. Dessa maneira uma árvore com três níveis gera definições que esperam três argumentos e produzem como resultado a expressão definida em *Notus* para a correspondente definição. Essa etapa é detalhada na Seção 5.8

A etapa **File writer** reúne as definições de funções e os *datatypes* que representam os domínios semânticos, e os escreve no módulo principal, *Main.hs*, do interpretador gerado para a linguagem especificada.

5.2 Compilação dos Domínios Semânticos

Os domínios semânticos definidos são compilados para *datatypes* em *Haskell*. Cada tipo de expressão de domínio em *Notus* (Seção 3.4.1) é traduzido para um *datatype* em *Haskell*. No entanto, traduções de alguns tipos de expressões de domínios podem gerar *datatypes* auxiliares. Essas traduções são detalhadas a seguir. Identificadores únicos são gerados pelo compilador sucedidos por um número seqüencial, como por exemplo os construtores de dados `E2` e `Enum0Green`, na Figura 5.2. Adicionalmente, os nomes dos módulos em que os identificadores são originalmente definidos, na especificação de uma linguagem, são concatenados aos nomes dos identificadores. Para simplificar a leitura dos *datatypes*, seus nomes e os nomes de seus construtores foram editados, eliminado-se os nomes dos módulos e separadores "__" utilizados entre o nome do módulo e os nomes originais.

Domínios Enumeração. Um domínio enumeração é traduzido para um componente no *datatype* do domínio semântico analisado. Esse valor mapeia para um *datatype* auxiliar, que possui um componente para cada elemento da enumeração em *Notus*. A Figura 5.2 mostra os *datatypes* criados para um domínio `Color` composto por uma enumeração.

```
Color = {red,green,blue};
```

```
data Color = E2 Enum0

data Enum0 = Enum0Green
           | Enum0Red
           | Enum0Blue
```

Figura 5.2: Exemplo de criação de *datatype* para enumeração de domínios.

O *datatype* `Color` representa o domínio `Color` que, nesse exemplo, é composto por apenas uma enumeração. O *datatype* `Enum0` representa a enumeração e seus construtores representam os enumerandos `red`, `green` e `blue`.

Domínios União Disjunta. Domínios uniões disjuntas são diretamente traduzidas para *datatypes* em *Haskell*, sendo cada elemento da união disjunta traduzido para um componente no *datatype* do domínio analisado. Cada componente gerado é composto por um construtor de tipos, seguido pelos valores de cada elemento da união disjunta. A Figura 5.3 mostra a geração do *datatype* para o domínio `Value`.

```
Value = Int | Bool;
```

```
data Value = U4 Int
           | U5 Bool
```

Figura 5.3: Exemplo de criação de *datatype* para união disjunta de domínios.

Domínios Produto Cartesiano. Domínios produto cartesiano são traduzidos para tuplas em *Haskell*, sendo um componente de *datatype* criado para representá-las. Como as tuplas em *Notus* possuem sempre um rótulo, esse é usado para a criação do construtor do componente do *datatype* criado. Assim, o componente criado é formado pelo rótulo da tupla e pela tupla. A Figura 5.4 mostra a construção do *datatype* para o domínio `Pair`, composto apenas por uma tupla, e o domínio `AtMostThree` composto por uma união disjunta de tuplas de diferentes tamanhos.

```
Pair = (Int,Bool);

AtMostThree =
    None ()
  | One (Int)
  | Two (Int,Int)
  | Three (Int,Int,Int);
```

```
data Pair = T6Pair (Int,Bool)

data AtMostThree =
    T7None ()
  | T8One (Int)
  | T9Two (Int,Int)
  | T10Three (Int,Int,Int)
```

Figura 5.4: Exemplo de criação de *datatype* para tupla de domínios.

Note que, como o domínio `Pair` é composto apenas por uma tupla, a declaração de um construtor para tupla é opcional, e nesse caso, `Pair` é usado como construtor.

Domínios de Listas. Domínios de listas geram componentes no *datatype* criado para o domínio analisado por meio da transformação do operador de lista de *Notus* `"*"` nos símbolos `"["` e `"]"` de *Haskell*, além da geração de um construtor de tipos para o novo valor. A Figura 5.5 mostra a geração do *datatype* para o domínio `A`, formado pela união disjunta de uma lista de valores inteiros, e uma lista de listas de valores booleanos, e para o domínio `B`, formado por uma lista de valores inteiros ou booleanos.

Para o domínio `A` a geração do *datatype* correspondente é realizada de forma direta. Já para o domínio `B` cria-se um *datatype* auxiliar, denominado `UD13B`, representando a união disjunta dos domínios `Int` e `Bool`. No *datatype* `B` é então criado um valor contendo uma lista de `UD13B`.

```
A = Int* | Bool**;

B = (Int|Bool)*;
```

```
data A = L11 [Int]
       | L12 [[Bool]]

data B = L16 [UD13B]

data UD13B = U14 Int
           | U15 Bool
```

Figura 5.5: Exemplo de criação de *datatype* para lista de domínios.

Domínio Funcional. Domínios funcionais são traduzidos para *datatypes* com componentes funcionais em *Haskell*. A Figura 5.6 mostra a geração do *datatype* para o domínio `State` e `ECont`.

```
State =
  Loc -> (Value|{error});

ECont =
  Value -> State -> State;
```

```
data State = F18 (Loc ->UD15State)

data UD15State = U16 Value
               | E17 Enum1

data Enum1 = Enum1Error

data ECont =
  F19 (Value -> (State -> State))
```

Figura 5.6: Exemplo de criação de *datatype* para união disjunta de domínios.

A tradução do domínio `State` produz três *datatypes*: o *datatype* `State` que representa o próprio domínio `State`, e os *datatypes* auxiliares `UD15State` e `Enum1`. O *datatype* auxiliar `UD15State` gerado representa a união disjunta de `Value` e da enumeração {`error`}, e o *datatype* auxiliar `Enum1` é devido à compilação da enumeração `error`.

A tradução do domínio `ECont` é direta, gerando apenas o *datatype* `ECont`, e como o operador $\rightarrow$ em *Notus* e associativo à direita, parênteses representando essa associatividade são gerados.

5.2.1 Construção do Grafo de Domínios G_d.

Após a compilação dos domínios semânticos para *datatypes* em *Haskell*, o compilador gera o grafo de domínios $G_d = (V_d, E_d)$ que representa a relação entre os domínios

definidos. Cada vértice $v \in V_d$ representa um domínio, e cada aresta $e = (u, v) \in E_d$, onde $u, v \in V_d$ indica que o domínio v compõe o domínio u. O grafo G_d representa a relação de subtipo entre os domínios definidos, uma vez que os domínios presentes na sua expressão são seus subtipos. A Figura 5.7 ilustra a criação do grafo de domínios para um conjunto de definições de domínios semânticos [1].

```
A = B | C;
B = {one, two};
C = Int;
D = E -> F;
F = G -> H;
E = I | J;
G = K | L;
```

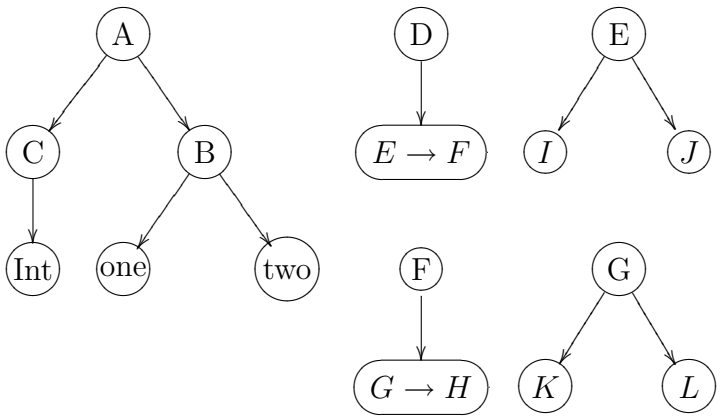

Figura 5.7: Exemplo de criação do grafo de domínios para um conjunto de domínios semânticos.

Após a criação do grafo de domínios G_d, o compilador de *Notus* verifica se existem definições ambíguas de domínio. Definições ambíguas ocorrem quando, para dois domínios `A` e `B` quaisquer em G_d, `A` alcança `B` por mais de um caminho. Quando a ambiguidade de domínios ocorre em *Notus*, um erro de compilação é gerado. O problema ocorre devido ao fato de que, dado um elemento `b` do domínio `B`, em um contexto onde o domínio `A` é esperado, não é possível definir qual caminho considerar entre `A` e `B` durante a geração de código (veja Seção 5.8). Os métodos identifyAmbiguities e visitToFindAmbiguities mostram como o compilador detecta a existência de ambiguidades no grafo G'_d.

O procedimento identifyAmbiguities calcula, na Linha 2, o conjunto base [2] para o grafo G_d. A seguir, para cada vértice v pertencente à base, o procedimento visitToFindAmbiguities realiza uma busca em profundidade [Sedgewick, 1990] verificando se v pode alcançar algum vértice u de mais de uma maneira. Quando isso

[1] Apesar dos elementos da enumeração `one` e `two` não serem domínios, *datatypes* são gerados para sua representação, e, portanto, esses vértices são necessários para a verificação de compatibilidade de tipos e geração de código.

[2] A base de um grafo é um subconjunto $B \subseteq G(V)$ tal que $\forall v, v \in B$ não há caminho entre u e v, e todo $w \notin B$ pode ser alcançado a partir de algum vértice de B.

```
identifyAmbiguities(G_d)
1  gBase ← calculateGraphBase(G_d)
2  for each v ∈ gBase
3       do initVisitedNodes(V[G_d])
4          ambiguities ← ∅
5          visitToFindAmbiguities(v,ambiguities)
6          if length[ambiguities] > 0
7             then ERROR("Ambiguities problem for domains " + ambiguities)
```

```
visitToFindAmbiguities(v, problems)
1  visitedNodes[v] ← true
2  adjs ← getAdjacenciesList(v)
3  for each u ∈ adjs
4       do if ¬visitedNodes [u]
5             then visitToFindAmbiguities(u,ambiguities)
6             else  ambiguities ← ambiguities + u
```

ocorre u é inserido em um conjunto *ambiguities*. O conjunto *ambiguities* é usado pelo compilador para reportar as ambiguidades encontradas. A Figura 5.8 exemplifica a ocorrência de definições ambíguas, e o grafo de domínios gerado G_d. Os domínios `A` e `String` são ambíguos, pois existem dois caminhos entre `A` e `String`: `A` → `String` e `A` → `B` → `C` → `String` .

```
A = Int | String | B;
B = C;
C = String;
```

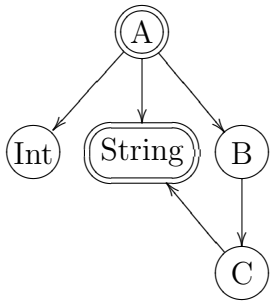

Figura 5.8: Exemplo de definições de domínios ambíguas, e o grafo de domínios G_d gerado. Os vértices destacados representam os domínios conflitantes.

Após verificada a não-existência de ambiguidades no grafo de domínios G_d, um novo grafo G'_d é criado, a partir do fecho transitivo direto do grafo G_d. O objetivo do grafo G'_d é auxiliar na verificação de tipos das expressões de definições de funções e na

etapa de geração de código. O grafo G'_d representa a relação de compatibilidade entre domínios. Dessa forma, a partir de um elemento `a` de um domínio `A` é possível verificar onde esse elemento pode ser usado, observando-se as relações do domínio `A` com os demais domínios no grafo G'_d. A Figura 5.9 mostra o grafo G'_d gerado a partir do grafo G_d da Figura 5.7. A arestas tracejadas representam as arestas incluídas no grafo G_d pelo cálculo do fecho transitivo direto. Observando o grafo G'_d é possível determinar que o elemento `int` do domínio `Int` pode ser usado onde um elemento do domínio `A` é esperado, já que existe uma aresta do domínio `A` para o domínio `Int` no grafo G'_d.

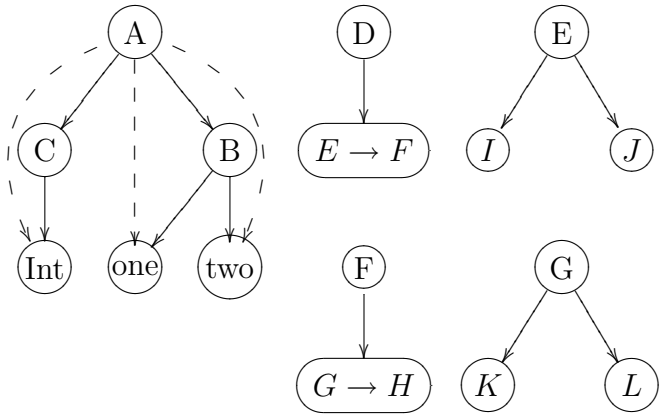

Figura 5.9: Grafo G'_d resultante do cálculo do fecho transitivo direto no grafo de domínios G_d da Figura 5.7.

5.3 Geração de Código Para *Datatypes*

Os *datatypes* gerados a partir dos domínios semânticos (veja Seção 5.2) são escritos no programa principal do interpretador gerado pelo compilador de *Notus* para a linguagem especificada. A expressão de atualização de função `update`, definida na Listagem 5.26, utiliza o operador de igualdade (==) e, portanto, a classe **Eq** de *Haskell* é obrigatoriamente mencionada na assinatura da função `update`. Para que a função `update` possa ser aplicada aos tipos do interpretador gerado, esses devem incluir uma cláusula de derivação da classe **Eq** em suas declarações.

Instâncias da classe **Eq** podem ser derivadas para *datatypes* básicos, com exceção de *datatypes* funcionais e de **IO** [Jones, 2003, Capítulo 6]. Dessa maneira, antes da geração de cada *datatype dt*, o compilador verifica se *dt* possui um componente funcional, ou se *dt* é composto por algum outro *datatype* que possui um componente funcional. Essa verificação é auxiliada pelo grafo de domínios G'_d. Os *datatypes* adjacentes a domínios funcionais não possuem a cláusula **deriving Eq** no código *Haskell* gerado, os demais

incluem essa cláusula em sua declaração. Considere o exemplo da Figura 5.10, onde são mostrados domínios semânticos definidos em *Notus*, e o grafo de domínios G'_d correspondente.

```
A = B | C;
B = D -> E;
C = String;
D = Int;
E = Double;
```

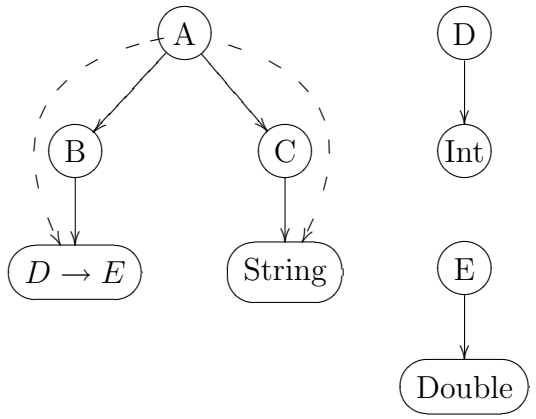

Figura 5.10: Domínios semânticos em Notus e o grafo G'_d correspondente gerado.

A Listagem 5.1 mostra os *datatypes* gerados para os domínios da Figura 5.10. O *datatype* `B` foi gerado sem a cláusula **deriving Eq**, pois possui um elemento funcional. O datatype `A` também não inclui essa cláusula, por ser adjacente, no grafo G'_d, ao domínio funcional `D -> E`. Os demais *datatypes* são gerados com a cláusula **deriving Eq**.

```
1  data A = U0 B
2         | U1 C
3
4  data B = F2  (D -> E)
5
6  data C = D3 String
7           deriving Eq
8
9  data D = D4 Int
10          deriving Eq
11
12 data E = D5 Double
13          deriving Eq
```

Listagem 5.1: *Datatypes* gerados para os domínios definidos na Figura 5.10

5.4 Explicitação de Parâmetros das Definições de Funções

Notus possibilita que definições de uma função sejam escritas com diferentes números de parâmetros, permitindo ao projetista ocultar, por motivos de simplicidade, argumentos que não são usados em uma determinada definição, como ocorre nas definições def_1 e def_2 da função `fun` na Listagem 5.2. Isso é comum em definições de funções semânticas de avaliações de construtos de uma linguagem, como por exemplo, a omissão na definição para uma construção que não utiliza *environments* e *store*. Em *Haskell* isso não é permitido, pois as definições de uma função devem possuir o mesmo número de parâmetros.

As definições de funções em *Notus*, com número de parâmetros diferentes, têm seus números de parâmetros igualados, para que possam ser analisadas pela etapa de compilação de **Detecção de Definições de Funções Sobrepostas**, que processa as definições duas a duas, comparando os padrões que as compõem. Para exemplificar como a explicitação de parâmetros ocorre, considere as definições para a função `fun` da Listagem 5.2. Os domínios semânticos `A` e `D` estão definidos na Figura 5.7. As expressões de cada definição da função `fun` estão representadas por $R_1 \cdots R_5$, já que essas não são relevantes para a transformação realizada nesta etapa.

```
function fun : A -> D;
fun a     = R1;  //def1
fun b e   = R2;  //def2
fun c i g = R3;  //def3
fun c i k = R4;  //def4
fun c e k = R5;  //def5
```

Listagem 5.2: Definições para a função `fun`.

Para explicitar os parâmetros das definições, identificadores artificiais, representando elementos dos domínios da declaração da função, são inseridos nas definições. Na Listagem 5.2 as definições 3, 4 e 5 são as que possuem maior número de parâmetros explícitos e, portanto, as definições de `fun` com número de parâmetros menor que 3 serão alteradas. Utilizando a definição dos domínios da Figura 5.7, o compilador expande os domínios funcionais `D` e `F`, como mostra a seqüência de passos a seguir, obtendo o domínio $A \rightarrow E \rightarrow G \rightarrow H$ para a função `fun`.

$A \rightarrow \mathbf{D}$; (válido pois $D = E \rightarrow F$)

$A \rightarrow E \rightarrow \mathbf{F}$; (válido pois $F = G \rightarrow H$)

$A \rightarrow E \rightarrow G \rightarrow H$

A Listagem 5.3 mostra as definições para função `fun` após a explicitação dos parâmetros. As inclusões de identificadores artificiais[3] realizadas foram: os elementos `e0` ∈ `E` e `g1` ∈ `G` na definição *def1*, e o elemento `g2` ∈ `G` na definição *def2*.

```
function fun : A -> E -> G -> H;
fun a e0 g1 = R1 e0 g1; //def 1
fun b e  g2 = R2 g2;    //def 2
fun c i  g  = R3;       //def 3
fun c i  k  = R4;       //def 4
fun c e  k  = R5;       //def 5
```

Listagem 5.3: Definições para a função `fun` após a explicitação dos parâmetros.

É importante observar que, para preservar a semântica das definições originais, os identificadores artificiais criados são aplicados às expressões do lado direito nas novas definições criadas.

5.5 Verificação de Tipos das Expressões das Equações Semânticas

Cada expressão em *Notus* possui um tipo associado. A verificação de tipos consiste em avaliar o tipo de cada expressão, de acordo com o tipo esperado para ela. Note que, em *Notus*, tipos são domínios. A compatibilidade entre tipos t_1 e t_2, em *Notus* ocorre quando o tipo t_1 pode ser usado onde t_2 é esperado. As situações em que um tipo t_1 é dito compatível com um tipo t_2 ocorrem se $t_1 = t_2$, ou se t_1 é subtipo de t_2. Diz-se que t_1 é subtipo de t_2 se existir uma aresta de t_2 para t_1 no grafo de domínios G'_d, condição válida para todos os tipos de domínios existentes em *Notus* (veja Seção 5.2), exceto para os casos em que t_1 e t_2 são tipos funcionais. Para t_1 e t_2 funcionais, usa-se a regra da contra-variância [Meyer, 1997, Cardelli e Wegner, 1986]: considerando que t_1 é um tipo na forma `A -> B`, e t_2 na forma `C -> D`, então t_1 é subtipo de t_2 se `C` é igual ou subtipo de `A` e `B` é igual ou subtipo `D`.

Além da verificação de compatibilidade do tipo esperado com o tipo da expressão, para cada uma das expressões existentes em *Notus*, algumas verificações padrões de tipos de seus componentes também são realizadas, como, por exemplo, para uma expressão `if` e_1 `then` e_2 `else` e_3 é preciso verificar se a expressão de teste do `if`, e_1, possui o tipo booleano, e se as expressões e_2 e e_3 possuem tipos compatíveis.

[3]Os identificadores artificiais únicos são gerados sucedidos por um número seqüencial.

Os padrões, ou seja, os identificadores que compõem as expressões de cada definição, ou são elementos de um domínio existente, ou são identificadores não-declarados[4]. O tipo dos identificadores não-declarado pode ser inferido pela assinatura da função. Por exemplo, considere o domínio `Value` e a função `hun` da Listagem 5.4. Os padrões da definição def_1, `int` e `string`, são declarados como elementos dos domínios `Int` e `String`. Já na definição def_2, os padrões `v` e `s` não pertencem, a princípio, a nenhum domínio, mas pela assinatura da função `hun` considera-se que, `v` pertence ao domínio `Value`, e `s` ao domínio `String`.

```
Value = Int | Double;

function hun : Value -> String -> Int;
hun int string = 1; //def 1
hun v   s      = 0; //def 2
```

Listagem 5.4: Determinação dos tipos dos padrões presentes no lado esquerdo das definições de funções.

Os identificadores provenientes do lado esquerdo da definição sempre possuem tipos pré-determinados, e, quando aparecem no lado direito, seu tipo pode ser recuperado e utilizado para a verificação de tipos da expressão de que fazem parte. No entanto, as expressões *let*, *where* e *abstração lambda* introduzem novos padrões, que, por sua vez, podem ser definidos como elementos de domínios existentes, ou podem ser identificadores não-declarados. Nesse caso, os identificadores não-declarados têm seus tipos inferidos, também pelo uso durante o processo de inferência de tipos.

Para que a verificação de tipos possa ser realizada de maneira uniforme, todo padrão deve possuir um tipo no momento de sua declaração, ou seja, antes do seu primeiro uso. Portanto, um tipo polimórfico é criado para cada um dos padrões, representados por identificadores não-declarados, das expressões *let*, *where* e *abstração lambda*. O tipo polimórfico para um padrão `p` possui um tipo instância que representará o tipo determinado para `p` durante o processo de verificação de tipos. O tipo polimórfico é unificado no seu primeiro uso, sendo então instanciado pelo tipo esperado no contexto em que é usado pela primeira vez e, a partir desse momento, o tipo polimórfico passa a ser representado pela sua instância.

A Listagem 5.5 mostra exemplos de expressões de *abstração lambda* e *let* com padrões não-declarados. O tipo da variável `x` da *abstração lambda* na definição da função `iun` tem seu tipo polimórfico unificado com `Int` durante a verificação de tipos

[4]Um identificador é dito não-declarado quando não se encontra, na especificação, um domínio cujo nome é o identificador com sua primeira letra capitalizada.

da operação binária '+'. Na função `jun`, os padrões `s` e `t` têm seus tipos unificados com o domínio `Value` na verificação de tipos da aplicação de função `lun s t`.

```
Value = Int | Double;

function iun : Int -> Int;
iun = \x -> x + 1;

function jun : Value;
jun = let {s = t ; t = s} in lun s t;

function lun : Value -> Value -> Value;
lun v1 v2 = v1;
```

Listagem 5.5: Determinação dos tipos para padrões em expressões de *abstração lambda*, e *let*.

Antes de se verificar o tipo de uma expressão, é preciso assegurar que todos os padrões que a compõem possuem um tipo inicial. Assim, uma varredura é realizada inicialmente na expressão em busca de novos padrões definidos por expressões *let*, *where* e *abstração lambda*. Um novo tipo polimórfico é criado para cada novo padrão representado pelo identificador sem domínio definido encontrado, e o padrão acompanhado de seu tipo é armazenado em uma lista denominada *polyPatternList*. O objetivo de *polyPatternList* é, além de armazenar os novos padrões e seus tipos, controlar o número de chamadas ao verificador de tipos, determinando o fim da análise de tipos. Esse controle é necessário porque nem sempre é possível unificar todos os tipos polimórficos de uma expressão em apenas uma chamada ao verificador de tipos.

O algoritmo VERIFYTYPE controla a verificação de tipos. Uma lista L global, acessível por VERIFYTYPE e pelos verificadores de tipos de cada expressão, é criada contendo inicialmente *polyPatternList*. A lista L é formada por listas com as configurações possíveis dos padrões das expressões *let*, *where* e *abstração lambda* e seus tipos, existentes durante o processo de verificação de tipos. Além disso, VERIFYTYPE recebe como parâmetros a expressão *exp* a ser avaliada, e o tipo t esperado para essa expressão, onde t é o contradomínio da definição de função a que *exp* pertence.

Para ilustrar o funcionamento da verificação de tipos para expressões em *Notus* que demandam mais de uma chamada ao verificador, considere a função `mun` da Listagem 5.6.

```
verifyType(exp, t)
1  for each l_i ∈ L
2       do viable ← checkType(exp,t,l_i)
3          if ¬ viable
4            then L ← L − l_i
5  if length[L] ≠ 1
6     then ERROR("Could not determine type for expression "++ exp)
```

```
function mun : (Value->String)->Value->String;
mun = \x y -> x y;
```

Listagem 5.6: Determinação dos tipos para padrões em expressões de *abstração lambda*, e *let*.

Para se determinar o tipo da expressão da definição de `mun` são necessárias três chamadas ao verificador de tipos. A configuração da lista *polyPatternList*, para cada uma das chamadas, é mostrada a seguir. Nessa lista, a_i representa um tipo polimórfico.

- Chamada 1: $< x : a_1 \ , \ y : a_2 >$;
- Chamada 2: $< x : A \rightarrow B \ , \ y : a_2 >$;
- Chamada 3: $< x : A \rightarrow B \ , \ y : A >$.

São mostrados, a seguir, os algoritmos de verificação de tipos para cada uma das expressões de *Notus*, e como esses verificadores manipulam a lista *polyPatternList*, e a lista global L. Exemplos ilustrando o funcionamento dos algoritmos são mostrados logo após sua apresentação. O procedimento compatible é usado pelos verificadores de tipo para verificar se os tipos passados como parâmetros são compatíveis, de acordo com os critérios de compatibilidade de tipos descrito no início desta seção. O procedimento getSuperType($\{t_1, t_2, \cdots, t_n\}$), onde o parâmetro $\{t_1, t_2, \cdots, t_n\}$ é um conjunto de tipos, é usado para recuperar o tipo mais abrangente t' entre os elementos do conjunto argumento. O tipo mais abrangente é o domínio que representa o menor limite superior[5] na hierarquia de tipos de uma descrição, como mostra a Figura 5.11. Adicionalmente, toda expressão e possui um tipo associado, acessível pelo comando *type*$[e]$. Para tipos funcionais, é possível recuperar seu domínio, por meio do comando *domain*[*type*$[e]$], e o contra-domínio por meio de *coDomain*[*type*$[e]$].

[5]Do inglês: *least upper bound*

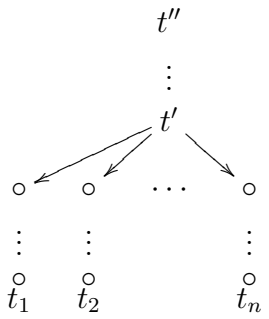

Figura 5.11: Tipo t' abrangente.

Verificador de Tipos Para Expressão *Literal*. Os literais de *Notus* são elementos dos domínios pré-definidos, listados na Seção 3.3. Assim, na verificação de tipos para literais, realiza-se a verificação da compatibilidade entre o tipo do literal e o tipo esperado t, como mostrado na Figura 5.12. Se o tipo do literal é compatível com o tipo esperado, então a verificação é bem sucedida, caso contrário, ocorre um erro de tipos.

```
checkType(exp, t, l)
1  if exp = literal
2     then if compatible(type[literal],t)
3             then type[exp] ← type[literal]
4             else return false
5          return true
   ▷ ( ··· )
```

Figura 5.12: Algoritmo de verificação de tipos para expressão *Literal*

A Listagem 5.7 mostra um exemplo de expressão literal, cujo tipo esperado é `Value`. Na verificação de tipo para a expressão `0`, é verificada a compatibilidade entre o tipo do literal, `Int`, e o tipo esperado `Value`. Como `Int` é subtipo de `Value`, o tipo resultante da expressão é o tipo do literal.

```
Value = Int | Bool;

function fun : Int -> Value;
fun 0 = 0;
```

Listagem 5.7: Exemplo de expressão *Literal*.

Verificação de Tipos Para Expressão *Identificador* O verificador de tipos para identificadores determina inicialmente se o identificador é um elemento da lista *polyPatternList*. Se o identificador é um padrão de expressão *let*, ou *where*, ou *abstração lambda*, ou seja, se o identificador pertencer à lista l, mas seu tipo em l não for compatível com o tipo t esperado, uma nova lista l' é criada a partir de l na Linha 13, substituindo-se o tipo do identificador pelo tipo esperado, e o algoritmo retorna FALSE indicando que a verificação de tipos não foi bem sucedida, o que causa a remoção da lista l de L pelo procedimento VERIFYTYPE. O algoritmo para verificação de tipos para expressão identificador é mostrado na Figura 5.13. A verificação de tipos para expressão identificador decorado é semelhante à realizada para expressão identificador, e, nesta verificação, o tipo esperado e o tipo do identificador decorado são um tipo lista.

```
CHECKTYPE(exp, t, l)
  ▷ ( ··· )
 6  if exp = identifier
 7     then if identifier ∉ l
 8             then if COMPATIBLE(type[identifier],t)
 9                     then type[exp] ← type[identifier]
10                     else return FALSE
11             else if COMPATIBLE(type[identifier],t)
12                     then type[exp] ← type[identifier]
13                     else l' ← l (type[identifier] ← t)
14                          L ← L + l'
15                          return FALSE
16          return TRUE
    ▷ ( ··· )
```

Figura 5.13: Algoritmo de verificação de tipos para expressão *Identificador*

A Listagem 5.8 mostra um exemplo de expressão identificador, cujo tipo esperado é `Value`. Na verificação de tipo para a expressão `x`, testa-se inicialmente se o identificador pertence à lista l, o que ocorreria se x fosse um componente de uma expressão *let*, ou *where*, ou *abstração lambda*. Nesse exemplo $x \notin l$, e, portanto, ocorre a verificação da compatibilidade entre o tipo `Int` do identificador, determinado pelo domínio da função, e o tipo esperado `Value`. Como `Int` é subtipo de `Value`, o tipo resultante da expressão é o tipo do identificador.

Verificação de Tipos Para Expressão *Operação Unária*. A verificação de tipos para operação unária recupera, a partir do operador *op*, os tipos funcionais em que

```
Value  = Int | Bool;

function fun : Int -> Value;
fun x = x;
```

Listagem 5.8: Exemplo de expressão *Identificador*.

op pode ser usado, por meio da chamada ao procedimento OPERANDTYPELIST(*op*). [6] Em seguida, testa-se, para cada um dos tipos funcionais possíveis $t_1 \rightarrow t_2$, se é possível avaliar o operando e_1 com tipo t_1, por meio da chamada a CHECKTYPE(e_1,t_1,l), e se os tipos t_2 e t são compatíveis. Se isso ocorrer, o tipo da expressão *exp* é t_2. O algoritmo para verificação de tipos para operação unária é mostrado na Figura 5.14.

```
CHECKTYPE(exp, t, l)
  ▷ ( ··· )
17  if exp = [op e1]
18     then ts ← OPERANDTYPELIST(op)
19          viable ← FALSE
20          for each (t1 → t2) ∈ ts
21               do if CHECKTYPE(e1,t1,l) ∧ COMPATIBLE(t2,t)
22                    then viable ← TRUE
23                         type[exp] ← t2
24          return viable
    ▷ ( ··· )
```

Figura 5.14: Algoritmo de verificação de tipos para expressão *Operação Unária*

A Listagem 5.9 mostra um exemplo de operação unária de negação aritmética, cujo tipo esperado é `Int`. A verificação de tipo para a expressão `- int` recupera os tipos funcionais permitidos por *Notus* para o operador `-`, por meio da chamada ao procedimento OPERANDTYPELIST(`-`), obtendo a lista ts = (`Int` → `Int`, `Double` → `Double`). Verificam-se então o tipo esperado e o tipo do operando `int`, `Int`, com cada um dos tipos funcionais da lista ts. O primeiro tipo funcional da lista, `Int` → `Int`, é compatível com os tipos do operando e com o tipo esperado. Dessa forma, o tipo da expressão unária `- int` é `Int`.

Verificação de Tipos Para Expressão *Operação Binária*. A verificação de tipos para operação binária ($[e_1\ op\ e_2]$) é semelhante à realizada para operação unária,

[6] Os tipos funcionais dos operadores (unários e binários) são definidos pela linguagem *Notus*, e se encontram representados internamente no compilador por uma tabela, acessada pelo procedimento OPERANDTYPELIST, que mapeia os operadores nos tipos funcionais esperados para estes.

```
function fun : Int -> Int;
fun int = - int;
```

Listagem 5.9: Exemplo de expressão *Unária*.

exceto que o tipo funcional para o operador binário tem a forma $t_1 \rightarrow t_2 \rightarrow t_3$. Realiza-se então a unificação de e_1 com o tipo t_1, e de e_2 com o tipo t_2. O algoritmo para verificação de tipos para expressão de operação binária é mostrado na Figura 5.15.

```
checkType(exp, t, l)
  ▷ ( ··· )
25  if exp = [e1 op e2]
26     then ts ← operandTypeList(op)
27          viable ← false
28          for each (t1 → t2 → t3) ∈ ts
29               do if checkType(e1,t1,l) ∧ checkType(e2,t2,l) ∧ compatible(t3,t)
30                    then viable ← true
31                         type[exp] ← t3
32          return viable
    ▷ ( ··· )
```

Figura 5.15: Algoritmo de verificação de tipos para expressão *Operação Binária*

A Listagem 5.10 mostra um exemplo de operação binária de adição aritmética, cujo tipo esperado é `Value`. A verificação de tipo para a expressão `x + y` recupera os tipos funcionais permitidos por *Notus* para o operador `+`, por meio da chamada ao procedimento operandTypeList(`+`) obtendo a lista ts = (`Int` → `Int` → `Int`, `Double` → `Double` → `Double`, `Float` → `Float` → `Float`). Verificam-se então os tipos dos operandos `x:Int, y:Int` e do tipo esperado, `Value`, com cada um dos tipos funcionais da lista ts. O primeiro tipo funcional da lista, `Int` → `Int` → `Int`, é compatível com os tipos dos operandos e com o tipo esperado, já que `Int` é subtipo de `Value`. Dessa forma, o tipo da expressão binária `x + y` é `Int`.

```
Value  = Int | Bool;

function fun : Int -> Int -> Value;
fun x y = x + y;
```

Listagem 5.10: Exemplo de expressão *Binária*.

Verificação de Tipos Para Expressão *If*. A verificação de tipos para expressão *if* ([if e_1 then e_2 else e_3]) realiza as verificações padrões, e quando essas são bem sucedidas, a expressão *if* possui tipo $type[e_2]$, caso o tipo de e_2 seja o mesmo de e_3. Caso contrário, e_2 e e_3 possuem tipos distintos porém compatíveis com o tipo esperado t. Recuperar-se o tipo mais abrangente t' entre t, e_2 e e_3 a ser atribuído à expressão *if*. O algoritmo para verificação de tipos para expressão *if* é mostrado na Figura 5.16.

```
checkType(exp, t, l)
  ▷ ( ··· )
33  if exp = [if e1 then e2 else e3]
34     then
35          if checkType(e1,Bool,l) ∧ checkType(e2,t,l) ∧ checkType(e3,t,l)
36            then if type[e2] = type[e3]
37                    then type[exp] ← type[e2]
38                    else t' ← getSuperType({t,type[e2],type[e3]})
39                         type[exp] ← t'
40                 return TRUE
41            else return FALSE
    ▷ ( ··· )
```

Figura 5.16: Algoritmo de verificação de tipos para expressão *If*

A Listagem 5.11 mostra um exemplo de expressão *if* cujo tipo esperado é `Value`. A verificação de tipo para a expressão `if b then true else 1` inicialmente verifica o tipo da expressão `b` com `Bool`, e os tipos das expressões `true` e `1` com o tipo esperado `Value`. Como os tipos das expressões do *if* são diferentes, recupera-se o tipo t' mais abrangente entre o tipo esperado `Value`, e os tipos das expressões do comando *if*, `true:Bool` e `1:Int`. Dessa forma, o tipo da expressão `if b then true else 1` é `Value`.

```
Value = Int | Bool;

function fun : Bool -> Value;
fun b = if b then true else 1;
```

Listagem 5.11: Exemplo de expressão *If*.

Verificação de Tipos Para Expressão *Case*. A verificação de tipos para expressão *case* (case e_0 of $\{p_1 \rightarrow e_1; \cdots; p_n \rightarrow e_n\}$) inicialmente verifica se e_0 é um identificador pertencente à lista l, caso isso ocorra, o tipo de e_0 em l é usado nas verificações seguintes. Caso e_0 não pertença à lista l seu tipo é obtido por meio da chamada

ao procedimento checkType passando-se um novo tipo polimórfico. Em seguida, o procedimento analyzePatternType é chamado para cada padrão p_i das cláusulas do *case*, que verifica se o tipo de p_i é compatível com o tipo de e_0. Os tipos da expressões $e_1 \cdots e_n$ são também verificados, e estes devem ser compatíveis com t. Após todas as verificações, o tipo mais abrangente existente entre $\{t, type[e_1] \cdots type[e_n]\}$ é associado à expressão *case*. O algoritmo para verificação de tipos para expressão *case* é mostrado na Figura 5.17.

```
checkType(exp, t, l)
  ▷ ( ··· )
42  if exp = [case e0 of {p1 → e1; ···; pn → en}]
43     then viable ← TRUE
44          if e0 ∉ l
45             then viable ← checkType(e0,newPolymorphicType(),l)
46          if viable
47             then if ∀ i ∈ {1, ···, n} : analyzePatternType(pi,type[e0])
48                     then if ∀ i ∈ {1, ···, n} : checkType(ei,t,l)
49                             then type[exp] ← getSuperType({t, type[e1] ··· type[en]})
50                                  return TRUE
51          return FALSE
    ▷ ( ··· )
```

Figura 5.17: Algoritmo de verificação de tipos para expressão *Case*

A Listagem 5.12 mostra um exemplo de expressão *case*, cujo tipo esperado é `Value`. A variável `v`, na expressão `case v of int -> 1; bool -> false`, não pertence à lista l, portanto seu tipo, `Value`, é verificado e recuperado na Linha 45 do algoritmo na Figura 5.17. Em seguida, é verificada a compatibilidade entre os tipos dos padrões das cláusulas da expressão *case*, `int` e `bool`, e o tipo da variável `v`, `Value`. As expressões das cláusulas do *case*, `1` e `false`, têm seu tipo verificado com o tipo esperado, `Value`. O tipo mais abrangente entre o tipo esperado e os tipos das expressões `Int` e `Bool` define o tipo da expressão `case v of int -> 1; bool -> false`, que para este caso é `Value`.

```
Value = Int | Bool;

function fun : Value -> Value;
fun v = case v of {int -> 1; bool -> false};
```

Listagem 5.12: Exemplo de expressão *Case*.

Verificação de Tipos Para Expressão *Agregado Lista.* A verificação de tipos para agregado lista testa inicialmente se o tipo esperado possui a forma de um tipo lista $t'*$ e, caso isso ocorra, o tipo de cada elemento do agregado lista [“(” $e_1 \cdots e_n$ “)”] deve ser verificado com o tipo esperado para os elementos de t'. Em seguida, o tipo mais abrangente dentre os elementos do agregado lista é recuperado, e o tipo resultante de *exp* é um tipo lista deste tipo. O algoritmo para verificação de tipos para expressão agregado lista é mostrado na Figura 5.18.

```
checkType(exp, t, l)
  ▷ ( ··· )
52  if exp = [“(” e1 ··· en “)”]
53     then viable ← TRUE
54          if t = t'*
55            then if ∀ i ∈ {1, ··· , n} : checkType(ei,t',l)
56                    then type[exp] ← (getSuperType({t‘, type[e1] ··· type[en]}))*
57                    else viable ← FALSE
58            else viable ← FALSE
59          return viable
    ▷ ( ··· )
```

Figura 5.18: Algoritmo de verificação de tipos para expressão *Agregado Lista*

A Listagem 5.13 mostra um exemplo de expressão agregado lista, cujo tipo esperado é `Value*`. Em seguida, os elementos do agregado lista `(int,bool)` têm seus tipos verificados com o tipo `Value`. Os elementos `int:Int` e `bool:Bool` são subtipos de `Value`, e portanto o tipo da expressão agregado lista é uma lista do tipo mais abrangente dentre seus elementos, `Value*`.

```
Value  = Int | Bool;

function fun : Int -> Bool -> Value*;
fun int bool = (int,bool);
```

Listagem 5.13: Exemplo de expressão *Agregado Lista.*

Verificação de Tipos Para Expressão *Agregado Tupla.* A verificação de tipos para tuplas inicialmente obtém o tipo tupla t' da tupla analisada, [id “(” $e_0, \cdots, e_n$ “)”], a partir de seu construtor id, e t' deve ser compatível com o tipo esperado t. O tipo tupla t' contém uma lista de tipos que correspondem aos tipos dos elementos da tupla. Cada elemento da tupla e_i tem seu tipo analisado de acordo com o tipo do elemento

correspondente na lista de tipos de t'. Em caso de sucesso nas verificações dos elementos da tupla, o tipo de *exp* é o tipo t' da tupla analisada. O algoritmo para verificação de tipos para expressão agregado lista é mostrado na Figura 5.19.

```
checkType(exp, t, l)
  ▷ ( ··· )
60  if exp = [id "(" e_0, ···, e_n ")"]
61     then t' ← findTupleType(id)
62          if compatible(t',t)
63             then if ∀ i ∈ {1, ···, n} : checkType(e_i,t_i,l)
64                     then type[exp] ← t'
65                          return true
66                     else return false
67          return false
    ▷ ( ··· )
```

Figura 5.19: Algoritmo de verificação de tipos para expressão *Agregado Tupla*

A Listagem 5.14 mostra um exemplo de expressão agregado tupla, cujo tipo esperado é `A`. A verificação de tipos para a expressão tupla `AT(i,s,b)` recupera seu tipo a partir de seu construtor `AT`, obtendo o tipo tupla `AT(Value,String,Bool)`. Esse tipo é subtipo do esperado `A`, então o tipo de cada elemento j da tupla (i,s,b) é analisado, de acordo com o tipo do *j-ésimo* elemento da tupla `AT(Value,String,Bool)`. Como todos os elementos das tuplas possuem tipos compatíveis com os tipos esperados, o tipo da expressão tupla é o tipo inicialmente obtido `AT(Value,String,Bool)`.

```
A = AT(Value, String, Bool);

function fun : Int -> String -> Bool -> A;
fun i s b = AT(i,s,b);
```

Listagem 5.14: Exemplo de expressão *Agregado Tupla*.

Verificação de Tipos Para Expressão *Concatenação de Listas*. A operação ++ é aplicável à expressões do tipo lista e à *strings*. Assim, as expressões e_1 e e_2 em e_1 ++ e_2, devem ser do tipo lista, representado no algoritmo a seguir por t'*, e os tipos dos elementos de e_1 e e_2 devem ser iguais. Além disso, o tipo esperado resultante da concatenação de duas listas também é um tipo lista. Em caso de sucesso nas verificações realizadas, o tipo de *exp* é $type[e_1]$. O algoritmo para verificação de tipos para expressão de concatenação de listas é mostrado na Figura 5.20.

```
checkType(exp, t, l)
  ▷ ( ··· )
68  if exp = [e1 ++ e2]
69     then
70          if t = t'*
71            then if checkType(e1,t'*,l) ∧ checkType(e2,t'*,l)
                              ∧ (type[e1] = type[e2])
72                      then type[exp] ← type[e1]
73                           return TRUE
74          return FALSE
    ▷ ( ··· )
```

Figura 5.20: Algoritmo de verificação de tipos para expressão *Concatenação de Listas*

A Listagem 5.15 mostra um exemplo de expressão de concatenação de listas, cujo tipo esperado é `Int*`. As listas da expressão de concatenação `l1 ++ l2`, têm seus tipos analisados com o tipo esperado `Int*`. Como os tipos de `l1:Int*` e `l2:Int*` são iguais, então o tipo da expressão de concatenação de listas é `Int*`.

```
function fun : Int* -> Int* -> Int*;
fun l1 l2 = l1 ++ l2;
```

Listagem 5.15: Exemplo de expressão *Concatenação de Listas*.

Verificação de Tipos Para Expressão *Construção de Listas*. A verificação de tipos para a construção de listas difere da realizada para concatenação de listas na verificação do tipo de e_1. Na construção de listas, o tipo de e_1 deve ser compatível com o tipo dos elementos de e_2, pois e_1 é um elemento a ser inserido na cabeça da lista e_2. O algoritmo para verificação de tipos para a expressão construção de listas é mostrado na Figura 5.21.

A Listagem 5.16 mostra um exemplo de expressão de construção de listas, cujo tipo esperado é `Value*`. A verificação de tipos para a expressão `i:l` analisa a compatibilidade entre o tipo da cabeça `i:Int` e `Value`, e entre o tipo da cauda `l:Int*` com `Value*`. Como os tipos da cabeça e da cauda da lista atendem aos tipos esperados, o tipo da expressão de construção de lista é `Value*`.

Verificação de Tipos Para Expressão *Let*. A verificação de tipos para expressão *let* (let { $p_1 = e_1$; $\cdots$; $p_n = e_n$ } in e_0) testa inicialmente se cada expressão e_i possui tipo compatível com o tipo de cada padrão p_i, correspondente. Em seguida, ocorre a

```
checkType(exp, t, l)
  ▷ ( ··· )
75  if exp = [e1 : e2]
76     then
77          if t = t'*
78            then if checkType(e1,t',l) ∧ checkType(e2,t'*,l)
79                              ∧ ((type[e1])* = type[e2])
80                      then type[exp] ← type[e2]
81                              return TRUE
82          return FALSE
    ▷ ( ··· )
```

Figura 5.21: Algoritmo de verificação de tipos para expressão *Construção de Listas*

```
Value = Int | Bool;

function fun : Int -> Value* -> Value*;
fun i l = i:l;
```

Listagem 5.16: Exemplo de expressão *Construção de Listas*.

verificação entre o tipo da expressão e_0 e o tipo t esperado. O algoritmo para verificação de tipos para expressão *let* é mostrado na Figura 5.22.

A verificação de tipos para expressão *where* é idêntica à realizada para expressões *let*, o que dispensa sua apresentação nesta seção.

```
checkType(exp, t, l)
  ▷ ( ··· )
83  if exp = [let {p1 = e1; ··· ; pn = en} in e0]
84     then if ∀ i ∈ {1, ··· , n} : checkType(ei,type[pi],l)
85             then if checkType(e0,t,l)
86                     then type[exp] ← type[e0]
87                             return TRUE
88          return FALSE
    ▷ ( ··· )
```

Figura 5.22: Algoritmo de verificação de tipos para expressão *Let*

A Listagem 5.17 mostra um exemplo de expressão *let* em uma definição para a função `fun`, cujo tipo esperado é `Value`. A função `gun`, utilizada na expressão *let*, também é apresentada. A verificação de lista para a expressão `let {int1 = gun v1;`

`int2 = gun v2} in int1 + int2` verifica se as expressões das cláusulas *let* são compatíveis com os tipos dos seu padrões. Por fim, o tipo da expressão do *let* `(int1 + int2):Int` deve ser compatível com o tipo esperado `Value`. Como `Int` é subtipo de `Value`, o tipo da expressão *case* é `Int`.

```
Value = Int | Bool;

function fun : Value -> Value -> Value;
fun v1 v2 = let {int1 = gun v1; int2 = gun v2} in int1 + int2;

function gun : Value -> Int;
gun bool = 0;
gun int = int;
```

Listagem 5.17: Exemplo de expressão *Let*.

Verificação de Tipos Para Expressão *Aplicação de Função*. A verificação de tipos de uma aplicação de função, $[e_1\ e_2]$, consiste em avaliar os tipos da função e_1 e do argumento e_2. O tipo esperado para a função e_1 pode ser obtido a partir do tipo esperado t, criando-se um tipo funcional t', onde o domínio, ainda desconhecido, é um tipo polimórfico novo, e o contradomínio é o próprio tipo t esperado como resultado da aplicação de função. A chamada ao verificador de tipos para a função e_1, avalia seu tipo de acordo com o tipo t'. O parâmetro da aplicação de função e_2 deve ser avaliado com o tipo do domínio da função e_1. E o resultado da aplicação de função é o contradomínio da função e_1. O algoritmo para verificação de tipos para expressão aplicação de função é mostrado na Figura 5.23.

```
checkType(exp, t, l)
  ▷ ( ··· )
89  if exp = [e1 e2]
90     then t' ← (newPolymorphicType() → t)
91          if checkType(e1,t',l) ∧ checkType(e2,domain[type[e1]],l)
92             then type[exp] ← coDomain[type[e1]]
93                  return TRUE
94          return FALSE
    ▷ ( ··· )
```

Figura 5.23: Algoritmo de verificação de tipos para expressão *Aplicação de Função*

A Listagem 5.18 mostra um exemplo de expressão aplicação de função, cujo tipo esperado é `Value`. O algoritmo de verificação de tipos para aplicação de função `gun i`

cria o tipo funcional $a \rightarrow$`Value` para a verificação do tipo da função `gun:Value->Bool`. O tipo `Value->Bool` é subtipo de $a \rightarrow$`Value` de acordo com a regra da contra-variância. A compatibilidade entre o tipo do parâmetro `i:Int` é avaliada com o tipo do domínio da função `gun`; esse teste é bem sucedido já que `Int` é subtipo de `Value`. O tipo da expressão é o contradomínio do tipo da função `gun`, `Bool`.

```
Value = Int | Bool;

function fun : Int -> Value;
fun i = gun i;

function gun : Value -> Bool;
gun int = true;
gun bool = bool;
```

Listagem 5.18: Exemplo de expressão *Aplicação de Função*.

Verificação de Tipos Para Expressão *Abstração Lambda*. A verificação de tipos para a *abstração lambda* testa, inicialmente, se o tipo esperado t é um tipo funcional ($t_1 \rightarrow t_2$). O tipo da variável de *abstração lambda* x deve possuir tipo compatível com t_1, e, caso isso ocorra, a expressão da *abstração lambda* e_0 é avaliada com o tipo t_2. O tipo de *exp* é um tipo funcional, com domínio do tipo de x e contradomínio do tipo de e_0. Caso o tipo da variável x não seja compatível com t_1, uma nova lista l' é criada a partir de l, substituindo-se o tipo de x por t_1, e o algoritmo retorna FALSE indicando que a verificação de tipos não foi bem sucedida, o que causa a remoção de l de L pelo procedimento VERIFYTYPE. A nova lista l' criada é usada para reavaliar a variável x, agora com tipo t_1, em uma chamada futura ao verificador de tipos como explicado no início desta seção. O algoritmo para verificação de tipos para *abstração lambda* é mostrado na Figura 5.24.

A Listagem 5.19 mostra um exemplo de expressão abstração lambda, cujo tipo esperado é `Value -> Int`. A variável da abstração lambda é um elemento do domínio `Int`, compatível com o tipo esperado `Value`. Em seguida, o tipo da expressão da abstração lambda, `int + 1` tem seu tipo verificado com o tipo esperado `Int`, o que é verdade. Então, o tipo da abstração lambda é o tipo funcional `Int -> Int`, formado pelos tipos da variável e da expressão da abstração lambda.

Verificação de Tipos Para Expressão *Composição de Função*. A verificação de tipos para composição de função, $[e_1 \bullet e_2]$ consiste em verificar os tipos das expressões e_1 e e_2 de acordo com o tipo do operador $\bullet : (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$. Sendo o tipo t esperado um tipo funcional na forma $t_1 \rightarrow t_2$, um novo tipo polimórfico

```
checkType(exp, t, l)
  ▷ ( ··· )
95  if exp = [\ x → e0]
96     then if t = t1 → t2
97             then if compatible(type[x],t1)
98                     then if checkType(e0,t2,l)
99                             then type[exp] ← (type[x] → type[e0])
100                                  return TRUE
101                    else l' ← l [type[x] ← t1]
102                            L ← L + l'
103                            return FALSE
104            return FALSE
    ▷ ( ··· )
```

Figura 5.24: Algoritmo de verificação de tipos para expressão *Abstração Lambda*

```
Value = Int | Bool;

function fun : Value -> Int;
fun = \int -> int + 1;
```

Listagem 5.19: Exemplo de expressão *Abstração Lambda*.

t_3, que representa a variável de tipo b na assinatura do operador $\bullet$, é obtido para auxiliar na verificação de e_1. Assim, a função e_1 é avaliada com o tipo $t_3 \rightarrow t_2$, e e_2 é avaliada com o tipo $t_1 \rightarrow domain[type[e_1]]$. Os tipos de e_1 e e_2 são usados na construção do tipo de *exp*, desde que as verificações dessas expressões sejam bem sucedidas. O algoritmo para verificação de tipos para expressão composição de função é mostrado na Figura 5.25.

A Listagem 5.20 mostra um exemplo de expressão composição de função, cujo tipo esperado é `String -> Int`. As funções da composição de função `gun` e `hun` também são apresentadas. A verificação para a expressão `gun . hun`, analisa o tipo de `gun:Value->Int` com o tipo esperado `a` → `Int`. Em seguida a compatibilidade do tipo de `hun:String->Int` é analisada com o tipo esperado `String` → `Value`, o que é verdade já que `String` → `Int` é subtipo de `String` → `Value` de acordo com a regra da contra-variância. O tipo da expressão é então formado pelo domínio de `hun` e o contradomínio de `gun`, `String` → `Int`.

Verificação de Tipos Para Expressão *Atualização de Função.* O tipo esperado para uma expressão de atualização de função, $[e_0$ "[" $e_1 \leftarrow e'_1, \cdots, e_n \leftarrow e'_n$ "]"$]$,

```
checkType(exp, t, l)
   ▷ ( ··· )
105  if exp = [e1 • e2]
106     then if t = t1 → t2
107             then t3 ← newPolymorphicType()
108                  if checkType(e1,(t3 → t2),l) ∧
                                checkType(e2,(t1 → domain[type[e1]],l))
109                    then type[exp] ← ((domain[type[e2]]) → (coDomain[type[e1]]))
110                         return true
111          return false
     ▷ ( ··· )
```

Figura 5.25: Algoritmo de verificação de tipos para expressão *Composição de Função*

```
function fun : String -> Int;
fun = gun . hun;

function gun : Value -> Int;
gun v = 0;

function hun : String -> Int;
hun d = 0;
```

Listagem 5.20: Exemplo de expressão *Composição de Função*.

deve ser um tipo funcional na forma $t_1 \rightarrow t_2$, correspondente ao tipo da função e_0. Cada cláusula $e_i \leftarrow e_i'$, pertencente a lista de atualização de função, deve possuir o lado esquerdo e_i compatível, de acordo com a regra da contra-variância utilizada para verificação de compatibilidade de tipos funcionais, com o domínio de $type[e_0]$. Já o lado direito, e_i', é analisado de maneira independente, pois este pode ser um supertipo do contradomínio de $type[e_0]$. Dessa maneira, o tipo de cada e_i' é analisado com o auxílio de um novo tipo polimórfico. Após a determinação do tipo de cada e_i' obtém-se o tipo mais geral, t_3, dentre os tipos de e_i', e o contradomínio do tipo funcional de e_0. O tipo da expressão de atualização de função é então determinado pelo tipo funcional $t_0 \rightarrow t_3$. O algoritmo para verificação de tipos para expressão de atualização de função é mostrado na Figura 5.26.

A Listagem 5.21 mostra um exemplo de expressão de atualização de função, cujo tipo esperado é `String -> Value`. A expressão `gun["a"<- 5,"b"<- false]` define novos valores em `gun` para os parâmetros *"a"* e *"b"*. Como o tipo de `gun` é compatível com o esperado, a verificação de tipos prossegue analisando os tipos das cláusulas da expressão de atualização. O lado esquerdo das cláusulas de atualização, *"a"* e *"b"*, devem ser

```
checkType(exp, t, l)
   ▷ ( ··· )
112  if exp = [e0 "[" e1 ← e'1, ··· , en ← e'n "]"]
113     then if t = t1 → t2
114             then if checkType(e0,(t1 → t2),l)
115                     then t0 ← domain[type[e0]]
116                          t'0 ← coDomain[type[e0]]
117                          if ∀ i ∈ {1, ··· , n} : checkType(ei,t0,l) ∧
                                    checkType(e'i,newPolymorphicType(),l)
118                            then t3 ← getSuperType({t'0, type[e'1] ··· type[e'n]})
119                                 type[exp] ← (t0 → t3)
120                                 return true
121          return false
     ▷ ( ··· )
```

Figura 5.26: Algoritmo de verificação de tipos para expressão *Atualização de Função*

compatíveis com o tipo `String`, e o lado direito, `5` e `false`, têm seus tipos obtidos a partir da análise com um novo tipo polimórfico. No fim, o tipo mais abrangente entre o contradomínio de `gun`, `Value`, e os tipos do lado direito das cláusulas de atualização de função, `Int` e `Bool`, é usado na formação do tipo para a expressão, que nesse caso é `String -> Value`.

```
function fun : String -> Value;
fun = gun["a" <- 5,"b" <- false];

function gun : String -> Value;
gun s = 0;
```

Listagem 5.21: Exemplo de expressão *Atualização de Função*.

Verificação de Tipos Para Expressão *Casamento de Padrão*. A verificação de tipos para expressão casamento de padrão,$[e_0 \text{ is } p]$, testa inicialmente se o tipo esperado para essa expressão é um tipo compatível com o tipo `Bool`, já que essa expressão sempre retorna verdadeiro ou falso de acordo com os elementos testados em seu corpo. Em seguida, o tipo de e_0 é obtido por meio de sua análise realizada com um novo tipo polimórfico, que é usado para verificar a aplicabilidade do padrão p ao tipo $type[e_0]$. O tipo de *exp* é `Bool`, desde que os testes tenham sido bem sucedidos. O algoritmo para verificação de tipos para expressão casamento de padrão é mostrado na Figura 5.27.

```
checkType(exp, t, l)
  ▷ ( ··· )
122  if exp = [e0 is p]
123     then if compatible(Bool,t)
124             then if checkType(e0,newPolymorphicType(),l) ∧
                            analyzePatternType(p,type[e0])
125                     then type[exp] ← Bool;
126                          return true
127          return false
     ▷ ( ··· )
```

Figura 5.27: Algoritmo de verificação de tipos para expressão *Casamento de Padrão*

A Listagem 5.22 mostra um exemplo de expressão de casamento de padrão, utilizada no teste de uma expressão `if`. A primeira verificação de tipo para a expressão `v is int` consiste em verificar se o seu tipo esperado é `Bool`, o que é verdade. Em seguida, o tipo de `v`, `Int` é obtido e usado para testar a aplicabilidade do padrão `int`. Como `Int` é subtipo de `Value`, o tipo da expressão é `Bool`.

```
function fun : Value -> Value;
fun v = if v is int then v else 0;
```

Listagem 5.22: Exemplo de expressão *Casamento de Padrão*.

Verificação de Tipos Para Expressão *Nodo de AST*. A verificação de tipos para expressões *nodo de AST* consiste apenas em verificar se o tipo do nodo é compatível com o tipo esperado, e caso isso seja verdade, o tipo da expressão *exp* é o próprio tipo do nodo. O algoritmo para verificação de tipos para expressão de nodo de *AST* é mostrado na Figura 5.28.

A Listagem 5.23 mostra um exemplo de expressão nodo de *AST*, utilizada como parte da expressão de denotação do comando *while*. A semântica para o comando de repetição `while e do c` reavalia o próprio comando enquanto `e` resultar em *true*. O tipo de expressões nodo de *AST* é o tipo de sua variável de gramática. No exemplo, `["while" e c]` é do tipo `C`, e o tipo esperado para essa expressão é `C`, já que o nodo aparece na expressão de aplicação de função `dc ["while" e c]`.

```
checkType(exp, t, l)
  ▷ ( ··· )
128  if exp = grNode
129     then if compatible(type[grNode],t)
130             then type[exp] ← type[grNode]
131                  return true
132          return false
     ▷ ( ··· )
```

Figura 5.28: Algoritmo de verificação de tipos para expressão *Nodo de AST*

```
c ::= "while" exp "do" c : ["while" exp c]

function dc : C -> State -> State;
dc ["while" e c] state =
  let {v = de e state; b = getBool v} in
        if b
          then (dc ["while" e c] . dc c) state
               else state;
```

Listagem 5.23: Exemplo de expressão *Nodo de AST*.

5.6 Análise dos Padrões das Definições

Definições de funções em *Notus* podem ser escritas em módulos distintos, e o projetista não precisa definir sua ordem de geração. *Haskell* processa o casamento de padrões em todas as construções em que é usado de cima para baixo e da esquerda para a direita [Jones, 2003]. Assim, *Haskell* processa as definições de funções na ordem em que essas são definidas no código-fonte. Dessa forma, o compilador de *Notus* deve ordenar as definições de forma que a semântica da linguagem definida não seja prejudicada pela forma como compiladores e interpretadores *Haskell* as processam.

Seja $Defs = \{def_1, def_2, \cdots, def_n\}$ o conjunto das definições para uma função resultante da etapa de **Function definitions arity analyzer**, descrita na Seção 5.4. O conjunto de definições *Defs*, para cada função declarada na especificação, é processado nessa etapa com o objetivo de se definir a ordem correta no código *Haskell* gerado. As seqüências de padrões das definições são ordenadas duas-a-duas. Sejam então $def_a \in \text{Defs} = \{p_{a.1}, p_{a.2}, \cdots, p_{a.n}\}$ e $def_b \in \text{Defs} = \{p_{b.1}, p_{b.2}, \cdots, p_{b.n}\}$ duas definições com seus respectivos conjuntos de padrões, onde n é a aridade da função analisada. Cada par de padrão $p_{a.i}$, e $p_{b.i}$, $i = 1 \cdots n$, é analisado com o auxílio do grafo de domínios G'_d (veja Seção 5.2), produzindo um *string* de símbolos de entrada

para o autômato finito determinístico M, cuja a função de transição é ilustrada pelo grafo na Figura 5.29. A análise consiste em comparar os domínios D_1 e D_2, determinando a relação existente entre eles no grafo G'_d. O resultado da comparação é um símbolo pertencente ao alfabeto do AFD $M = (S, \Sigma, \delta, s_0, A)$ onde:

- S é o conjunto de estados do autômato $\{s_0, s_1, s_2, s_3, s_4\}$, onde:
  - s_0 indica que as duas definições são iguais;
  - s_1 indica que a definição def_a é mais específica do que a definição def_b;
  - s_2 indica que a definição def_b é mais específica do que a definição def_a;
  - s_3 indica que as definições def_a e def_b são conflitantes, ou seja, existe pelo menos uma seqüência de padrões para a função analisada que pode ser casada em ambas as definições;
  - s_4 indica que as duas definições def_a e def_b são disjuntas, ou seja, dada uma seqüência de padrões $pSeq$ para a função analisada, ou $pSeq$ casa com def_a, ou casa com def_b.
- Σ é o alfabeto formado por $\{e, f, s, n, d\}$, onde:
  - e é o símbolo que representa que os padrões $p_{a.i}$ e $p_{b.i}$ possuem o mesmo domínio (*equals*);
  - f é o símbolo que representa a existência de uma aresta no grafo de domínios G'_d de D_1 para D_2 (*first*);
  - s é o símbolo que representa a existência de uma aresta no grafo de domínios G'_d de D_2 para D_1 (*second*);
  - n é o símbolo que indica que, para o grafo G'_d, $\hat{\Gamma}^+(D_1) \cap \hat{\Gamma}^+(D_2) \neq \emptyset$, ou seja, os domínios D_1 e D_2 possuem subtipos em comum (*non-empty*);
  - d é o símbolo que indica que os domínios D_1 e D_2 são disjuntos, ou seja, não existe relação entre D_1 e D_2 no grafo G'_d (*disjoints*).
- δ é a função de transição definida pela tabela de transição de estados na Tabela 5.1.
- A é o conjunto de estados de aceitação do autômato $\{s_0, s_1, s_2, s_3, s_4\}$.

Os estados de aceitação do autômato M, obtidos para cada par de definições, são usados para a construção de um novo grafo, que define a ordem de geração das definições. O grafo construído é definido por $G_{def} = (V_2, E_2)$ onde cada vértice $v \in V_2$ representa a seqüência de padrões de uma definição, e cada aresta $e = (u, v) \in E_2$,

	e	f	s	d	n
s_0	s_0	s_1	s_2	s_4	s_3
s_1	s_1	s_1	s_3	s_4	s_3
s_2	s_2	s_3	s_2	s_4	s_3
s_3	s_3	s_3	s_3	s_4	s_3
s_4	s_4	s_4	s_4	s_4	s_4

Tabela 5.1: Tabela de transição de estados para o autômato M.

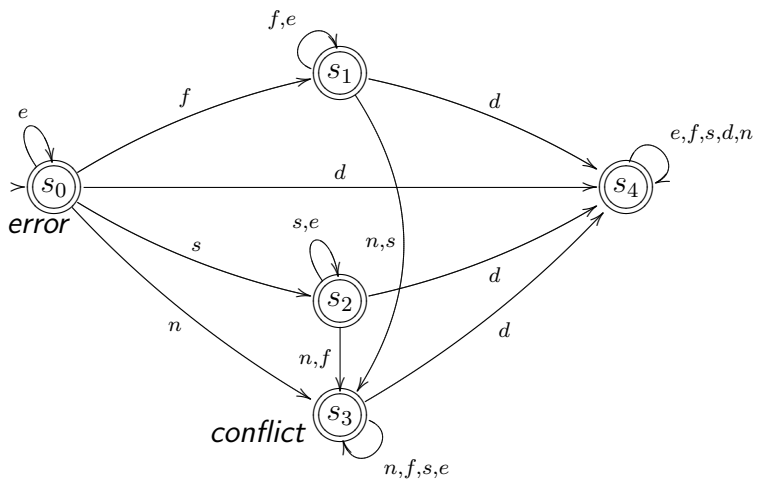

Figura 5.29: Representação gráfica da função de transição para o autômato M, usado na determinação da ordem de geração das definições de uma função.

onde $u, v \in V_2$ tal que $\hat{\delta}(s_0, \gamma(u,v)) = s_1$, e $\gamma(u,v) \in \Sigma^*$ é a seqüência de resultados da comparação dos domínios dos padrões de def_a e def_b, com base no grafo de domínios G'_d. De forma análoga, uma aresta $(v,u) \in E_2$ é inserida no grafo G_{def} se $\hat{\delta}(s_0, \gamma(u,v)) = s_2$. Sendo def_a representada por u e def_b por v, uma aresta (u,v) indica que def_a deve ser gerada antes de def_b, e uma aresta (v,u) indica que def_b deve ser gerada antes de def_a.

A ocorrência do estado de aceitação s_0 gera um erro de execução, pois indica que as definições analisadas são iguais e, portanto, devem ser refeitas pelo projetista. O estado de aceitação s_4 não gera restrição ao compilador e as definições analisadas, caso independentes, podem ser geradas em qualquer ordem. O estado de aceitação s_3 indica a ocorrência de um conflito entre def_a e def_b. O compilador avalia o conflito determinando, por meio das demais definições para a função analisada, se def_a e def_b podem ser consideradas disjuntas, ou se constituem um caso de erro. O conflito ocorre quando existe pelo menos uma seqüência de padrões de chamada que pode ser casada tanto com def_a quanto com def_b. No entanto, essa ambigüidade pode ser resolvida

pelas demais definições existentes, mas se não o for, o conflito se torna um erro. A análise feita para o caso de conflitos entre duas definições é composta pelos seguintes passos:

1. cálculo da interseção do $\hat{\Gamma}^+$ dos domínios de cada par de padrão $p_{a.i}$ com domínio $D_{a.i}$, e $p_{b.i}$ com domínio $D_{b.i}$, gerando uma lista de domínios: $< (\hat{\Gamma}^+(D_{a.1}) \cap \hat{\Gamma}^+(D_{b.1})), \cdots, (\hat{\Gamma}^+(D_{a.n}) \cap \hat{\Gamma}^+(D_{b.n})) >$;

2. combinação da lista do passo anterior gerando uma lista L_c de possibilidades de seqüência de padrões, que contém as seqüências de padrões que casam tanto com def_a quanto com def_b;

3. para cada uma das demais definições $def_j = \text{Defs} \setminus \{def_1, def_2\}$ com $j = 1 \cdots n$, existentes para a função analisada, verifica se def_j está presente na lista L_c. Caso isso ocorra, a seqüência de padrões de def_j é eliminada da lista L_c. Dessa maneira, o padrão que casaria tanto com def_a quanto com def_b é tratado por def_j.

4. Ao fim da análise de todas a definições def_j, se L_c estiver vazia, então def_a e def_b são disjuntas, pois nesse caso todos os padrões conflitantes foram resolvidos. Caso contrário, um erro é gerado pelo compilador.

A Figura 5.30 mostra o grafo gerado para as definições da função `fun` (definida na Listagem 5.2). Note que, os identificadores `e0,g1` e `g2`, que aparecem nas definições de `fun`, foram criados pelo compilador pela etapa de explicitação de parâmetros das definições da funções (veja Seção 5.4).

```
function fun : A -> E -> G -> H;
fun a e0 g1 = R1 e0 g1; //def 1
fun b e  g2 = R2 g2;    //def 2
fun c i  g  = R3;       //def 3
fun c i  k  = R4;       //def 4
fun c e  k  = R5;       //def 5
```

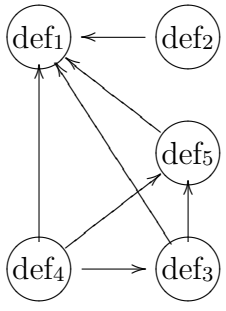

Figura 5.30: Grafo G_{def} de definição de função para as definições da função `fun`

As definições def_2, def_3, def_4 e def_5 estão contidas em def_1, pois o conjunto de padrões de def_1 é mais geral que os demais. Por exemplo, $\gamma(def_1, def_2) = \{s, e, e\}$, e,

```
function fun : A -> E -> G -> H;
fun b e  g2 = R2 g2;     //def 2
fun c i  k  = R4;        //def 4
fun c i  g  = R3;        //def 3
fun c e  k  = R5;        //def 5
fun a e0 g1 = R1 e0 g1; //def 1
```

Listagem 5.24: Definições para a função `fun` ordenadas.

portanto, $\hat{\delta}(s_0, \gamma(def_1, def_2)) = s_2$, e arestas são criadas de def_2, def_3, def_4 e def_5 para def_1 em G_{def}. De forma análoga, a definição def_4 é mais específica que as definições def_5 e def_3, e def_3 é mais específica que def_5, criando-se arestas entre essas definições.

Após a construção do grafo G_{def}, o compilador de *Notus* executa o algoritmo de ordenação topológica [Sedgewick, 1990] em G_{def}, obtendo a ordem de geração de funções. Um resultado da ordenação topológica para o grafo da Figura 5.30 é:
$< def_2, def_4, def_3, def_5, def_1 >$. A Listagem 5.24 mostra as definições na ordem em que são geradas para *Haskell*.

5.7 Detecção de Definições de Função Sobrepostas

A etapa de detecção de definições de funções que se sobrepõem gera um erro de compilação caso o compilador determine que uma dada definição def_a de uma função `fun` nunca é casada no conjunto de padrões de chamadas a `fun`. Isso ocorre quando def_a é coberta pelos padrões que a antecedem na ordem de geração em código *Haskell*. Por exemplo, considere a função `boolToInt` definida na Listagem 5.25. O padrão da definição def_3 nunca é casado, pois todos os valores possíveis para o domínio `Bool` já estão representados em def_1 e def_2, e dessa maneira, def_3 é dita ser sobreposta por def_1 e def_2.

```
function boolToInt : Bool -> Int;
boolToInt false = 0;  //def 1
boolToInt true  = 1;  //def 2
boolToInt bool  = 2;  //def 3
```

Listagem 5.25: A definição def_3 não é alcançável.

A função `boolToInt`, se submetida ao compilador *Haskell GHC*, gera uma mensagem de advertência, reportando que o padrão da definição def_3 é sobreposto. O compilador de *Notus* gera um erro de compilação caso detecte definições sobrepostas. Essa decisão tem o objetivo de gerar código *Haskell* livre de erros e advertências. Além disso, o

fato de as definições poderem ser escritas em módulos distintos pode acarretar na não percepção do projetista da linguagem da existência de sobreposição entre essas definições.

A detecção de definições sobrepostas é realizada por meio da inspeção, para cada vértice v do grafo G_{def}, se v é coberta $\hat{\Gamma}^{-}(v)\backslash\{v\}$, onde $\hat{\Gamma}^{-}(v)$ é o fecho transitivo inverso do vértice v[7]. Assim, se a definição representada pelo vértice v está completamente representada pelas definições que a antecedem na ordenação de geração, um erro de compilação é reportado pelo compilador de *Notus*.

O algoritmo de detecção de definições sobrepostas é dado pelo método recursivo overlappingFunDef, que é executado para cada definição de função da especificação. Os parâmetros do método overlappingFunDef são: uma definição de função v; o fecho transitivo inverso da definição v, $X = \hat{\Gamma}^{-}(v)\backslash\{v\}$; o índice i do padrão de v a ser analisado. O método overlappingFunDef inicialmente verifica se a definição v é igual a alguma definição de X, o que indica que a definição v é sobreposta por X. Caso isso não aconteça, o domínio do padrão v_i é comparado com as expressões de domínios compostas, que podem estabelecer uma relação de subtipagem: enumeração, união disjunta e o domínio `Bool`. Para esses casos, novas definições *case* são geradas, substituindo o tipo mais abrangente pelos subtipos que o compõem. Na Linha 16, as novas definições *case* são verificadas por overlappingFunDef e, se todas elas estiverem presentes em X, determina-se que v é sobreposta por X.

[7] O fecho transitivo inverso do vértice v é composto pelos vértices que o alcançam.

```
overlappingFunDef(v, X, i)
 1  if v ∈ X
 2     then return TRUE
 3     else  if v_i =Bool
 4              then cases = {< v_0 ··· v_{i−1} > •TRUE • {< v_{i+1} ··· v_{n−1} >,
 5                                  < v_0 ··· v_{i−1} > •FALSE • {< v_{i+1} ··· v_{n−1} >}
 6              else  if v_i = enum {e_1, e_2, ···, e_m}
 7                       then for each e_k
 8                                  do cases = cases + {< v_0 ··· v_{i−1} > •e_k • {< v_{i+1} ··· v_{n−1} >}
 9              else  if v_i =union{A_1 | A_2 | ··· | A_m}
10                       then for each e_k
11                                  do cases = cases + {< v_0 ··· v_{i−1} > •A_k • {< v_{i+1} ··· v_{n−1} >}
12              else  if i < (LENGTH(v))−1
13                       then return overlappingFunDef(v,X,i+1)
14              else
15                    return FALSE
16  for each case ∈ cases
17        do if overlappingFunDef(case,X,i+1) = FALSE
18              then return FALSE
19  return TRUE
```

Após a verificação de definições sobrepostas, o compilador gera uma árvore com as definições, onde cada nó é um padrão da definição. A altura da árvore é dada pela aridade da função. Assim, cada nível i da árvore é composto por padrões com domínios compatíveis com o i-ésimo domínio da declaração da função. A Figura 5.31 mostra a árvore construída a partir das definições ordenadas da Listagem 5.24.

5.8 Geração de Código Para Definições de Funções

A geração de código para definições de funções é dividida em duas etapas seqüenciais para cada definição de cada função: (a) geração do lado esquerdo da definição (parâmetros) e (b) geração do lado direito da definição da função (expressão).

5.8.1 Geração do Lado Esquerdo da Definição.

A geração do lado esquerdo das definições é guiada pela árvore de definições produzida pela etapa de detecção de definições sobrepostas. Para uniformizar a geração, as definições de funções escritas em *Notus* por meio de equações são transformadas em definições na forma de abstrações lambda, cujo corpo consiste em uma expressão **case**. Assim, cada parâmetro do lado esquerdo da definição de função em *Notus* é escrito

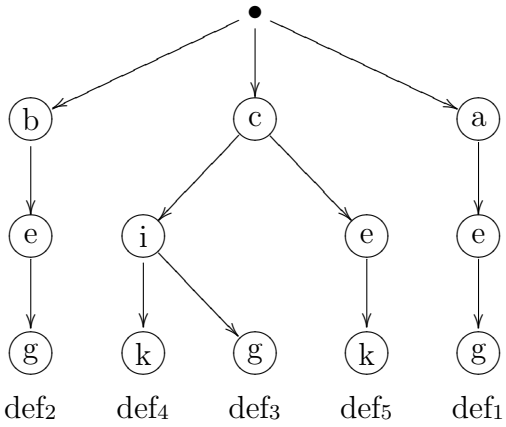

Figura 5.31: Árvore com as definições de função ordenadas para a geração para *Haskell*.

como um parâmetro de uma *abstração lambda* do lado direito da definição da função, no código *Haskell* gerado, de forma que a *abstração lambda* obtém o argumento de chamada da função para esse parâmetro. Os níveis da árvore são percorridos em profundidade da esquerda para a direita a partir da raiz. Para cada nodo gera-se uma *abstração lambda*. Se o nodo possui somente um filho, apenas a *abstração lambda* é gerada e esse filho é o próximo a ser analisado. Se o nodo possuir mais de um filho, suas ramificações são traduzidas para uma expressão *case*, e cada nodo filho é representado por uma cláusula dessa expressão. A expressão *case* gerada para um nodo x com n ramos é:

$$\begin{array}{lll} \backslash \mathsf{x} & \rightarrow & \mathbf{case}\ \mathsf{x}\ \mathbf{of}\ \{r_1 \rightarrow (\cdots); \\ & & \vdots \quad \rightarrow \ddots\ ; \\ & & r_n \rightarrow (\cdots)\} \end{array}$$

A *abstração lambda* gerada tem seu tipo t' comparado ao tipo correspondente da declaração da função t, e se t' é subtipo de t, então ocorre a injeção de tipo de t' para t.

Sempre que um identificador deve ser gerado, seja este pertencente ao lado esquerdo ou ao lado direito da definição de função original, seu tipo é comparado ao tipo esperado no contexto em que ele se encontra. Exitem três possíveis relações entre o tipo t' do identificador e o tipo esperado t: (1) t' é igual a t; (2) t' é subtipo de t e; (3) t' é supertipo de t. No caso (1) o identificador é gerado diretamente. No caso (2) o identificador deve ser injetado no tipo t, e no caso (3) o identificador é projetado do

tipo t. Um novo nome único é criado para representar o identificador, evitando conflitos de nomes no código gerado, e o novo nome artificial é mapeado ao original na definição a que pertence. Assim as ocorrências subsequentes do identificador original podem ser substituídas pelo novo nome.

Para que as injeções e projeções dos tipos t e t' ocorram, é preciso que se recupere o caminho entre os tipos t e t' no grafo de domínios G'_d. O caminho obtido contém os tipos existentes entre t e t', e é usado para criar um *string* formado pelos construtores de tipos dos *datatypes* que representam os tipos do caminho. O procedimento GENPATHFROMSUPERTOSUB percorre o caminho entre o supertipo t e o subtipo t', criando o *string pString* com os construtores de tipos. Para cada tipo u do caminho, recupera-se o elemento do *datatype* que o representa na Linha 4, em seguida, na Linha 5, o construtor de tipos do elemento é concatenado ao *string* em formação.

```
GENPATHFROMSUPERTOSUB(G'_d,t,t')
1  path ← GETPATH(G'_d,t,t')
2  currentData ← GETDATATYPE(t)
3  for each u ∈ path
4        do dataElement ← u[currentData]
5           pString ← pString + typeConstructor[dataElement]
6           currentData ← GETDATATYPE(u)
```

Para exemplificar como o *string* com os construtores de tipos é obtido, considere o exemplo da Figura 5.32, que apresenta um trecho de uma definição em *Notus* e o código *Haskell* gerado para esse trecho. Em *Notus* são definidos os tipos `A` e `C`, e uma função `pathTest : A -> String`, cuja definição tem como lado esquerdo o padrão `int`. No código *Haskell*, o parâmetro `int` é transformado em uma *abstração lambda* do lado direito da definição com o nome artificial `int20`, e o *string* que representa o caminho no grafo G'_d entre o tipo do parâmetro `Int` e o tipo esperado `A` é "`U6 D8`", observando-se que `U6` é o construtor de tipos criado para representar o subtipo `C` do tipo `A` em *Notus*, e `D8` é o construtor criado para o subtipo `Int` do tipo `C`.

5.8.2 Geração do Lado Direito da Definição.

Ao se alcançar uma folha na árvore de definições de uma função, todos os parâmetros para a definição analisada já foram gerados, e a expressão para essa função deve ser gerada. As expressões comuns a *Notus* e a *Haskell* têm sua geração realizada de forma direta. São elas: literal; identificador; *if*; *case*; agregado tupla; concatenação e cons-

```
A = B | C;
C = Int;

function pathTest : A -> String;
pathTest int = "ok";
```

```
data A = U5 B
       | U6 C
        deriving Eq

data C = D8 Int
        deriving Eq

pathTest :: (A -> String)
pathTest = (\(U6(D8 int20)) ->"ok")
```

Figura 5.32: Exemplo de injeção de tipos: o identificador `int20:Int` é injetado no tipo esperado `A`.

trução de listas; *let*; *where*[8]; aplicação de função; e *abstração lambda*. Os algoritmos de geração de código *Haskell* para as demais expressões são detalhados a seguir.

O algoritmo para geração de expressões é dado pelo procedimento genFunExp, que recebe como parâmetros a expressão a ser gerada e o tipo esperado para essa função. Como essa etapa é posterior à de verificação de tipos, o formato dos tipos esperados, quando relevantes, é escrito no próprio cabeçalho dos algoritmos apresentados.

Composição de função. No código gerado para composição de função, $[e_1 \bullet e_2]$, cria-se uma *abstração lambda* com variável l para representar o domínio do tipo funcional da composição t_1. Assim, a *abstração lambda* é gerada, seguida pela geração de e_1, e_2, e novamente l como argumento de e_2. O algoritmo para geração de código para composição de função é mostrado na Figura 5.33. O procedimento genString(s), concatena o *string* s ao código a ser gerado para expressão analisada, e o procedimento newIdentifier retorna um novo identificador artificial único.

A Figura 5.34 mostra a geração para uma expressão de composição de função. A definição para a função `comp` é formada pela composição das funções `compAB` e `compCA`.

Identificador e literal. A geração de código para as expressões identificadores e literais possui uma peculiaridade devido ao fato de que essas expressões geram injeções e projeções de tipos. O algoritmo para geração de código para expressões de identificador é mostrado na Figura 5.35. O tipo do identificador é comparado ao tipo esperado t, e caso esses tipos sejam diferentes, uma injeção ou projeção de tipo é gerada.

[8]Internamente o compilador de *Notus* trata expressões *where* como expressões *let*, já que essencialmente essas expressões são semanticamente equivalentes.

```
genFunExp(exp, t1 → t2
1  if exp = [e1 • e2]
2     then id ← newIdentifier()
3          genString("\"+id +" → ")
4          genFunExp(e1,(domain[type[e1]] →t2))
5          genString("(")
6          genFunExp(e2,(t1 → domain[type[e1]]))
7          genString(id)
8          genString(")")
   ▷ ( ··· )
```

Figura 5.33: Algoritmo de geração de código para expressão `Composição de Função`

```
function compAB : A -> B;
function compCA : C -> A;

function comp : C -> B;
comp = compAB . compCA;
```

```
comp :: (C -> B)
comp =  \x55-> compAB (compCA x55)
```

Figura 5.34: Exemplo de geração de código para expressão de composição de função.

```
genFunExp(exp, t)
  ▷ ( ··· )
 9  if exp = identifier
10     then if type[identifier] = t
11             then genFunExp(identifier)
12             else if isSubType(type[identifier],t)
13                    then injectionType(type[identifier],t)
14                    else projectionType(t,type[identifier])
    ▷ ( ··· )
```

Figura 5.35: Algoritmo de geração de código para expressão *Identificador*

Casamento de Padrão. A expressão casamento de padrão, $[e_0 \,\mathrm{is}\, p]$, é traduzida para uma expressão *case* em *Haskell*. A expressão do case é formada pelo *string* de e_0, e o corpo do *case* é gerado com duas cláusulas. A primeira contém o padrão `p`, e quando casada, retorna `True`. A segunda contém o padrão *don't care* (_), retornando `False` para qualquer outro argumento que não seja o padrão `p`. O algoritmo para

geração de código para expressões casamento de padrão é mostrado na Figura 5.36.

```
genFunExp(exp, t
  ▷ ( ··· )
15  if exp = [e0 is p]
16     then genString("case")
17          genFunExp(e0,type[e0])
18          genString("of{\n")
19          genFunExp(p,type[e0])
20          genString("→ True;\n")
21          genString(" (_) → False }")
    ▷ ( ··· )
```

Figura 5.36: Algoritmo de geração de código para expressão de *Casamento de Padrão*

A Figura 5.37 mostra a geração para uma expressão casamento de padrão, considerando as definições de domínios mostrados na Figura 5.7 na Página 111. A definição para a função `isPatternTest` utiliza uma expressão casamento de padrão para testar se o argumento passado é o padrão `one`, elemento da enumeração do domínio `B`.

```
B = {one,two};

function isPTest : B -> Int;
isPTest b = if b is one
              then 1
              else 0;
```

```
isPTest :: (B -> Int)
isPTest = \b137->
  if (case  b137 of{
       (E8(EnumOne)) -> True;
        _            -> False})
    then 1
    else 0
```

Figura 5.37: Exemplo de geração de código para expressão casamento de padrão.

Atualização de função. A semântica esperada para expressões de atualização de função em *Notus* é implementada pela função `update`, definida em *Haskell* na Listagem 5.26. A função `update` recebe como parâmetros uma função `f` do tipo `a -> b`, uma lista `l1` do tipo `[a]`, e uma lista `l2` do tipo `[b]`, e retorna uma função, que dado um argumento `y` do tipo `a`, percorre `l1` seqüencialmente procurando por `y`, e quando o encontra, retorna o elemento de `l2` correspondente à mesma posição de `y` em `l1`. Caso `y` não seja encontrado, retorna-se o valor definido para `y` na função `f`. Assim, a geração de código para expressões de atualização de função,$[e_0$ "[" $e_1 \leftarrow e'_1, \cdots, e_n \leftarrow e'_n$ "]"],

consiste em construir uma chamada à função `update`. O primeiro argumento da função `update` é obtido pela chamada ao procedimento GENFUNEXP para e_0, os argumentos seguintes são as duas listas formadas a partir das cláusulas $e_i \leftarrow e'_i$, sendo a primeira lista formada pelos elementos $e_1 \cdots e_n$, e a segunda por $e'_1 \cdots e'_n$. O algoritmo para geração de código para expressões de atualização de função é mostrado na Figura 5.38.

```
update :: Eq a => (a -> b) -> [a] -> [b] -> a -> b
update f [] [] = \y-> f y
update f (x:xs) (v:vs) = \y -> if y == x
                                 then v
                                 else (update f xs vs y)
```

Listagem 5.26: Função `update`

```
GENFUNEXP(exp, t1 → t2)
  ▷ ( ··· )
22  if exp = [e0 "[" e1 ← e'1, ··· , en ← e'n "]"]
23     then GENSTRING("(update")
24          GENFUNEXP(e0,t1 → t2)
25          GENSTRING("[")
26          ∀ i ∈ {1, ··· , n} : GENFUNEXP(ei,t1)
27          GENSTRING("]")
28          GENSTRING("[")
29          ∀ i ∈ {1, ··· , n} : GENFUNEXP(e'i,t2)
30          i ← i + 1
31          GENSTRING("[ )")
    ▷ ( ··· )
```

Figura 5.38: Algoritmo de geração de código para expressão *Atualização de Função*

As Figuras 5.39 e 5.40 mostram exemplos de geração de código para expressão de atualização de função. O primeiro exemplo apresenta essencialmente o código gerado para a atualização de função. A função `fun`, desse exemplo, atualiza a função `gun` para os parâmetros *"a"* e *"b"*, que passam a mapear os valores `1` e `2` respectivamente.

O exemplo da Figura 5.40 mostra o código gerado para a função `funUpdateTest`, que recebe como parâmetros uma lista de *strings*, outra de inteiros, e um estado, retornando também um estado. A função `funUpdateTest` utiliza a expressão de atualização para mapear os elementos da cabeça das listas no estado, recebido como argumento, gerando um novo estado acrescido do mapeamento realizado.

Nodo de *AST*. Um nodo de *AST* em *Notus* pode ser composto por expressão de identificador, de literal *string* entre aspas, de identificador decorado ou por outro

```
function fun : String -> Int;
fun = gun["a"<-1,"b"<-2];

function gun : String -> Int;
gun s = 0;
```

```
fun :: (String -> Int)
fun = (update gun ["a","b"] [1,2])
```

Figura 5.39: Exemplo de geração de código para expressão de atualização de função.

```
State = String -> Int;

function funUpdateTest : String* -> Int* -> State -> State;
funUpdateTest () () s = s;
funUpdateTest s:ss* n:ns* state =
  funUpdateTest ss* ns* state[s<-n];
```

```
data State = F20  (String -> Int)

funUpdateTest :: ([String] -> ([Int] -> (State -> State)))
funUpdateTest =
  (\string117-> case string117 of {
    ([]) ->(\([])->(\state120 ->state120 ));
    ((string122:string123)) ->
      (\(int126:int127) ->
        (\state128 ->((( funUpdateTest string123 ) int127 )
                      (F20 (update (case state128 of
                        ((F20 fun131)) -> fun131) [string122] [int126])
                      ))
         )
      )
})
```

Figura 5.40: Exemplo geração de código para expressão de atualização de função com projeção.

elemento nodo de *AST*. Assim a geração de código para expressão de nodo de *AST* apenas percorre sua lista de elementos, efetuando uma chamada ao algoritmo de geração de código para cada uma das expressões que o compõem.

Operações unária e binária. Alguns operadores de *Notus* são representados por símbolos diferentes dos usados em *Haskell* para uma mesma operação. Por exemplo, o operador unário em *Notus* para negação aritmética é "−", e, em *Haskell*, usa-se a

função `negate` para a mesma operação. Para tratar esses casos, o compilador de *Notus* mantém uma tabela que mapeia os operadores de *Notus* para seus correspondentes em *Haskell*. A geração de código para operações unárias e binárias acessa essa tabela e obtém o *string* do operador a ser gerado. Os operadores são gerados sempre na forma prefixada. A Figura 5.41 mostra o código em *Notus* e o código *Haskell* gerado para as operações de conjunção booleana e negação aritmética.

```
function opBin :
    Bool -> Bool -> Bool;
opBin bool1 bool2 =
    bool1 and bool2;

function opUna:Int->Int;
opUna int = -int;
```

```
opBin :: (Bool -> (Bool -> Bool))
opBin = (\bool97->( \bool98 ->
           bool97 && bool98))

opUna :: (Int -> Int)
opUna = (\int100-> negate  int100 )
```

Figura 5.41: Exemplo de geração de código para operações unárias e binárias.

Agregado Lista. A geração de código para agregado lista é realizada de forma direta. A única diferença entre o código *Notus* e o código *Haskell* gerado para essa expressão é a substituição dos delimitadores de lista de *Notus* "(" e ")", para seus correspondentes em *Haskell*, "[" e "]". Assim, a geração envolve apenas gerar o *string* "[", seguido por chamadas ao gerador de código para cada um dos elementos da lista, e, no fim, a geração de "]". A Figura 5.42 mostra o código em *Notus* e o código *Haskell* gerado para agregado lista.

```
function toList :
  Int -> Int -> Int*;
toList n1 n2 = (n1,n2);
```

```
toList :: (Int -> (Int -> [Int]))
toList = (\int103->
           (\int104 ->
             [int103,int104]))
```

Figura 5.42: Exemplo de geração de código para agregado lista.

5.9 Transmissão de Contexto

As construções para transmissão de contexto de *Notus* foram apresentadas na Seção 3.8. Essas construções foram definidas tardiamente e apenas o suporte para a sua futura

implementação foi incorporado ao compilador apresentando neste trabalho. A especificação sintática da linguagem *Notus* contém produções sintáticas para essas construções e foram criadas interfaces de implementação para as demais etapas, para permitir a extensão da versão atual de forma a suportar totalmente as novas construções. Assim, o processo de extensão do compilador consiste em implementar as fases de análise semântica e geração de código para essas novas construções.

5.10 Transformadores de Módulos

Transformador de módulos é uma construção de *Notus* que permite a extensibilidade e a escalabilidade nas definições da semântica de linguagens de programação. Por exemplo, uma especificação escrita em semântica denotacional direta pode ser transformada em uma especificação com semântica de continuações, por meio da inclusão de um único módulo, que define, via um transformador de módulos, como as equações existentes passam a ser tratadas. A versão atual do compilador *Notus*, apresentada neste trabalho, não implementa as construções de extensibilidade da especificação semântica, constituindo-se em trabalho futuro a extensão do compilador para suporte dessas construções.

A princípio, duas alternativas são vislumbradas para a incorporação da transformação de módulos na atual versão do compilador *Notus*. A primeira consiste em compilar os transformadores gerando um módulo *Haskell* capaz de transformar a *AST* do restante da especificação. A *AST* transformada seria novamente reinserida no compilador *Notus*. O problema dessa solução é estabelecer representações equivalentes da *AST* e formas de comunicação entre os nodos da *AST* no compilador *Notus* e os nodos da *AST* em *Haskell*. Uma segunda alternativa é implementar um interpretador para os transformadores de módulos de forma que seja possível alterar os nodos da *AST* no compilador *Notus*. A *AST* transformada seria então fornecida como entrada para o gerador de códigos. O problema dessa solução é o custo de se projetar e implementar um interpretador para esse fim.

As Figuras 5.43 e 5.44 esquematizam essas duas alternativas. Os processos e dados sombreados constituem a extensão a ser implementada. Existe uma interseção entre essas alternativas. A idéia comum a elas diz respeito à construção de 3 AST's: (1) *AST* contento os identificadores dos módulos a serem transformados e das funções transformadoras a serem aplicadas a esses módulos, obtida a partir das cláusulas de importação; (2) *AST* das funções transformadoras; e (3) o restante da *AST* da especificação. Além disso, nas duas alternativas é necessário que seja verificada a consistência das aplicações de transformação nos módulos de uma especificação a partir das informações do item (1)

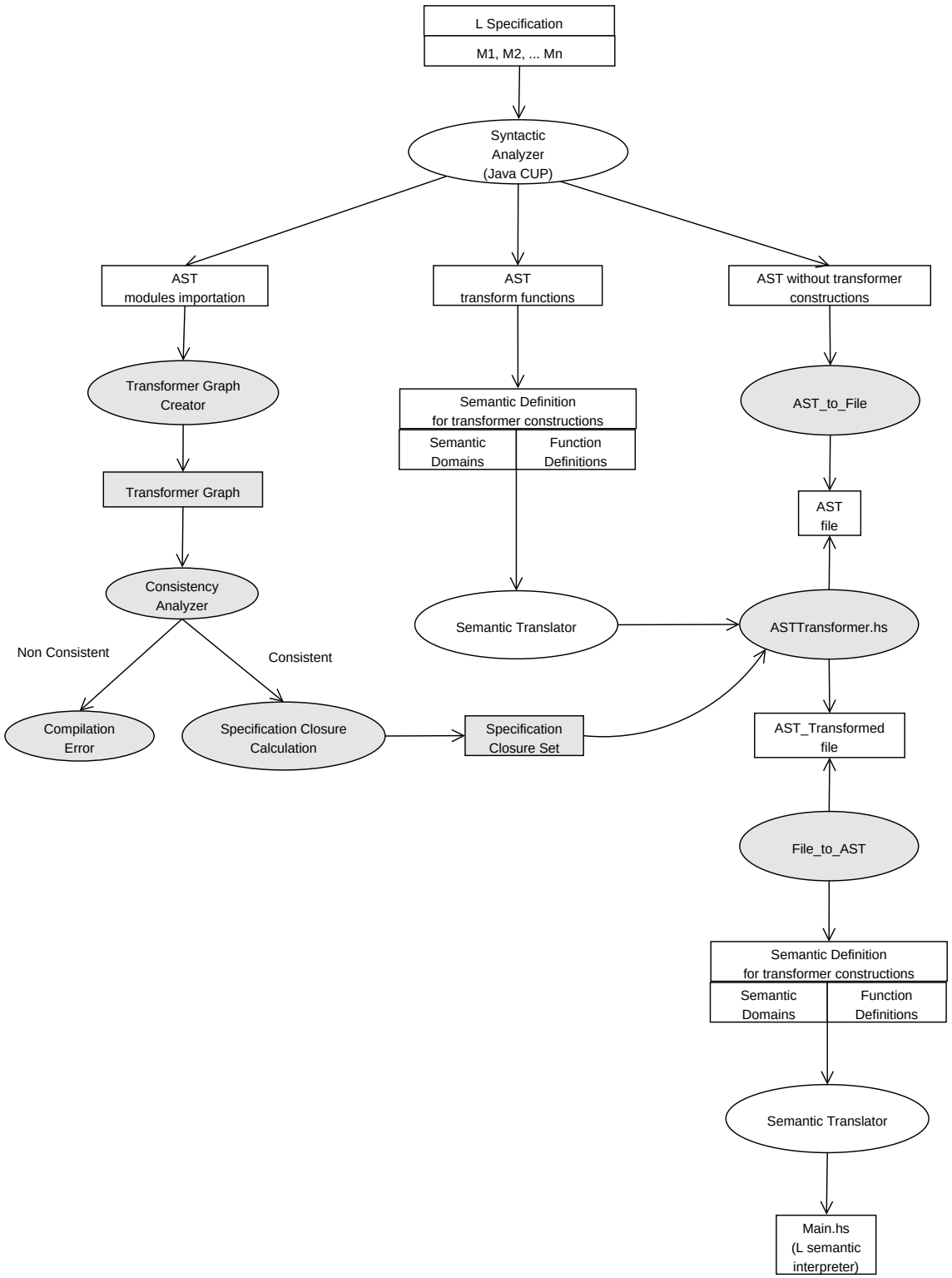

Figura 5.43: Diagrama esquemático da implementação de transformação de módulos utilizando um módulo *Haskell* transformador.

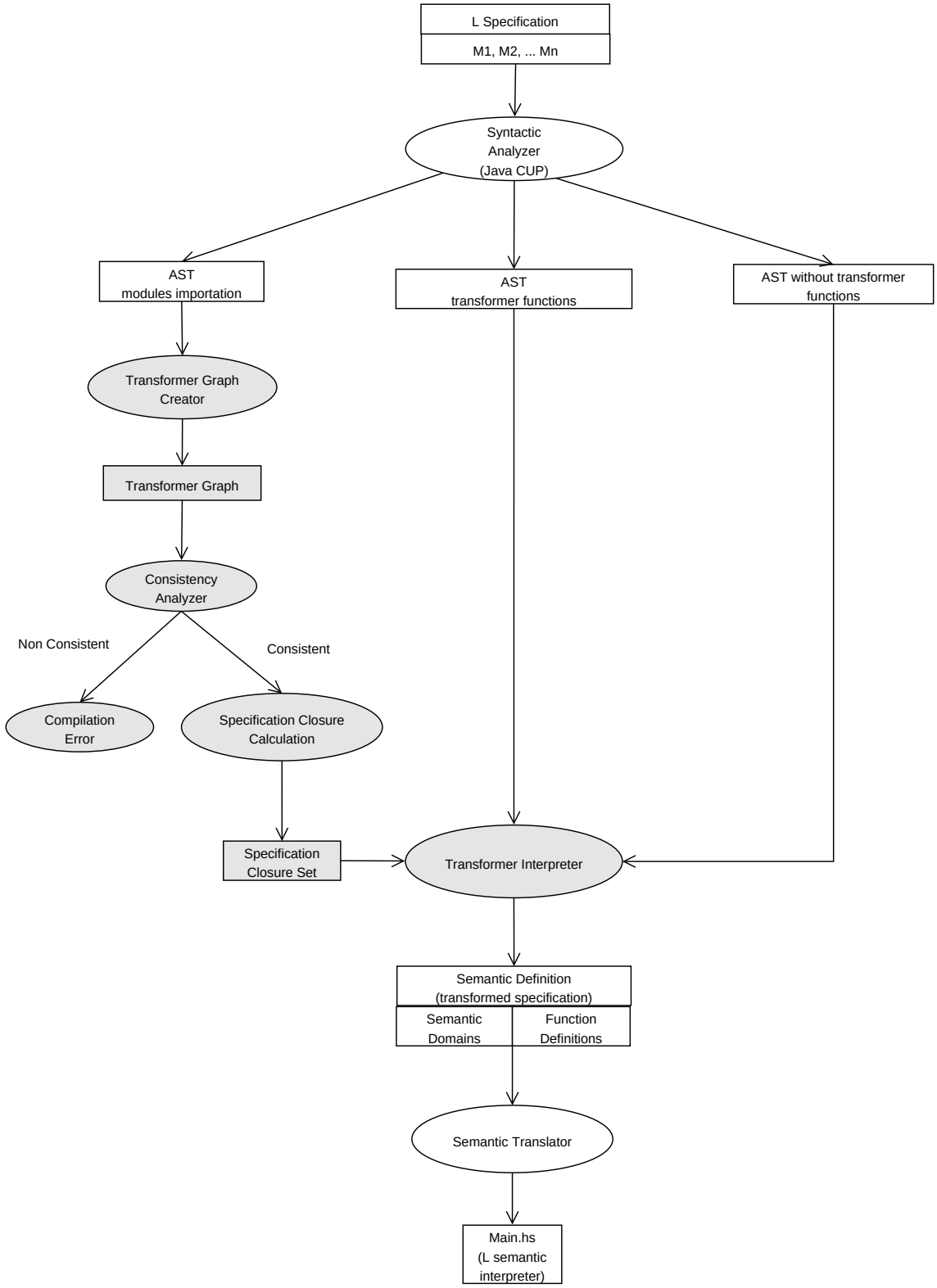

Figura 5.44: Diagrama esquemático da implementação de transformação de módulos utilizando um interpretador de transformadores.

supra-descrito. Nas alternativas apresentadas, as aplicações de transformação são modeladas por meio do grafo de aplicação de transformações, que é um grafo dirigido e rotulado $G = (V, E)$, tal que V é o conjunto dos módulos da especificação, e o conjunto de arestas é definido como $E = \{(m_1, m_2) \in V \times V \mid \text{o módulo } m_1 \text{ importa o módulo } m_2\}$. Cada aresta em E é rotulada pela função de transformação utilizada na importação, ou pela função identidade i, se a importação não definir transformação a ser aplicada. Por exemplo, o grafo de aplicação de transformações referente aos módulos da Seção 3.9 é o mostrado na Figura 5.45.

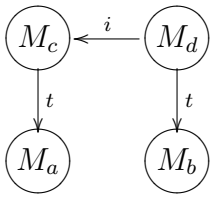

Figura 5.45: Grafo de transformação para a especificação da Figura 3.2 na Página 57

Os caminhos entre os vértices do grafo definem a sequência de aplicações de funções de transformação aos módulos do grafo. O grafo criado é analisado quanto à sua consistência. O teste da consistência verifica, para os caminhos não nulos existentes entre dois módulos, por exemplo entre m_1 e m_2, se esses caminhos são formados pela mesma seqüência de funções de transformação. Esta condição garante que a interpretação de cada elemento do módulo m_2 no módulo m_1 é unicamente definida. Após a verificação de consistência, gera-se o fecho da especificação que apresenta, para cada módulo, a seqüência de funções a serem aplicadas aos seus elementos. Por exemplo, para o grafo de transformação da Figura 5.45 em que o conjunto de módulos é dado por $M = \{M_a, M_b, M_c, M_d\}$, o fecho da especificação M' é o conjunto:

$$M' = \{(t)\ M_a, (t)\ M_b, M_c, M_d\}$$

O diagrama da Figura 5.43 apresenta de forma esquemática a compilação dos transformadores de módulos utilizando-se um módulo *Haskell* transformador da AST do restante da especificação. O módulo transformador é gerado a partir da compilação das funções de transformação da especificação. A AST do restante da especificação é gravada em arquivo, e o módulo transformador lê a AST e, em conjunto com o fecho de transformação, aplica as transformações na AST e armazena em arquivo a AST transformada. Essa AST é reinserida no compilador Notus, que então a traduz para o interpretador semântico da linguagem especificada.

O diagrama da Figura 5.43 apresenta a segunda alternativa descrita para a transformação de módulos, que consiste em interpretar as funções de transformação e alterar a *AST* da especificação no próprio compilador *Notus*. A *AST* contendo as funções de transformação, o fecho da transformação, e a *AST* contendo o restante da especificação são entradas para um módulo interpretador de transformação do compilador *Notus*. Esse interpretador transforma a *AST* da especificação de acordo com as informações recebidas e gera uma nova especificação transformada, que é então submetida ao módulo de tradução semântica do compilador *Notus* para a geração do interpretador da linguagem especificada.

As duas alternativas para a extensão do compilador *Notus* para suporte à transformação de módulos são, a princípio, viáveis, caracterizando-se como trabalho futuro a análise das vantagens e desvantagens de cada uma delas, para que se possa definir qual seguir.

5.11 Um Exemplo Completo: Linguagem *LEE*

Esta seção apresenta um exemplo completo em *Notus*, contemplando as especificações léxica, sintática e semântica de uma linguagem de expressões simples. A especificação de *LEE*, quando compilada por *Notus*, gera um interpretador de *LEE*, e é formada apenas pelo módulo principal mostrado na Listagem 5.27.

5.11.1 Definição Léxica de *LEE*

A definição léxica de *LEE* é composta por:

- *token* `num` pertencente ao domínio `Exp`, cujo lexema é um número inteiro;
- macro `digit` representado um dígito entre 0 e 9;
- operadores "+"e "*";
- delimitadores "("e ")".

5.11.2 Sintaxe de *LEE*

A gramática concreta de *LEE* é $G_l = \{V_l^c, T_l^c, R_l^c, \texttt{exp}\}$, onde:

- $V_l^c = \{\texttt{exp , term , factor}\}$;
- $T_l^c = \{\texttt{num , + , * , (,)}\}$;

```
1  module Main
2
3  syntax exp;
4  semantics dProg;
5
6  //-------------------------Lex Definition
7
8  token num : Exp = digit+ is asInteger;
9  element digit = [0-9] ;
10
11 //-------------------------Syn Definition
12
13 exp ::= exp "+" term
14       | term : term;
15
16 term : Exp ::= term "*" factor
17              | factor : factor ;
18
19 factor : Exp ::= num : num
20                | "(" exp ")" :exp;
21
22 //-------------------------Sem Definition
23
24 function dProg : Exp -> String;
25 dProg exp = integerToString (dExp exp);
26
27 function dExp : Exp -> Int;
28 dExp int = int;
29 dExp [exp1 "+" exp2] = dExp exp1 + dExp exp2;
30 dExp [exp1 "*" exp2] = dExp exp1 * dExp exp2;
31 end
```

Listagem 5.27: Especificação em *Notus* para uma linguagem de expressões.

- R_l^c é o conjunto que contém as regras:

$$
\begin{array}{lcl}
\text{exp} & \rightarrow & \text{exp "+" term} \\
& | & \text{term} \\
\text{term} & \rightarrow & \text{term "*" factor} \\
& | & \text{factor} \\
\text{factor} & \rightarrow & \text{num} \\
& | & \text{"(" exp ")"}
\end{array}
$$

5.11.3 Gramática Abstrata de *LEE*

A gramática abstrata de *LEE* é $G_l = \{V_l^a, T_l^a, R_l^a, \texttt{Exp}\}$, onde:

- $V_l^a = \{\texttt{Exp}\}$;
- $T_l^a = \{\texttt{+ , *}\}$;
- R_l^a é o conjunto que contém as regras:

```
Exp  →  Exp "+" Exp
     |  Exp "*" Exp
     |  Int
```

5.11.4 Domínios de *LEE*

LEE é composta apenas pelo domínio `Exp` definido por:

```
Exp = Int + (Exp × \token{+} × Exp) + (Exp × \token{*} × Exp)
```

5.11.5 Interpretador de *LEE*

O interpretador de *LEE* compreende os analisadores léxico e sintático e o programa principal. As listagens contendo os arquivos gerados pelo compilador de *Notus* para a especificação de *LEE* são mostradas a seguir:

5.11.5.1 Analisador Léxico.

O analisador léxico é produzido pelo gerador de analisadores léxicos *Alex* a partir do arquivo de especificação léxica *Lexer.x* gerado pelo compilador de *Notus*. O arquivo *Lexer.x* é mostrado na Listagem 5.28.

5.11.5.2 Analisador Sintático.

O analisador sintático é produzido pelo gerador de analisadores sintáticos *Happy* a partir do arquivo de especificação sintática *Syntactic.y* gerado pelo compilador de *Notus*. O arquivo *Syntactic.y* é mostrado na Listagem 5.29.

5.11.5.3 Gramática Abstrata

A gramática abstrata é representada por *datatypes* e é utilizada, tanto pelos analisadores léxico e sintático, quanto pelo programa principal. Assim, o compilador de *Notus* gera um módulo denominado `DataModule`, contendo os *datatypes* da gramática abstrata,

```
1  {
2  module Lexer where
3  import NotusDefault
4  import DataModule
5  import NotusFunctions
6  }
7
8  %wrapper "basic"
9
10 @digit = [0-9]
11
12 tokens :-
13
14 "("              { \s->(T__token2)}
15 "+"              { \s->(T__token0)}
16 @digit+          { \s->(T__num(Exp4 (asInteger s)))}
17 ")"              { \s->(T__token3)}
18 "*"              { \s->(T__token1)}
19
20
21
22 {
23 -- The token type:
24
25 data Token = T__token0
26            | T__token1
27            | T__token2
28            | T__token3
29            | T__num Exp
30         deriving Eq
31 }
```

Listagem 5.28: Arquivo de especificação léxica *Lexer.x* para o *Alex*

e os analisadores léxico e sintático e o programa principal importam esse módulo. A Listagem 5.30 mostra o módulo `DataModule` gerado para a especificação de *LEE*.

5.11.5.4 Programa Principal

O programa principal é composto pelas funções semânticas da especificação de *LEE* e pela função principal `main`, que inicia a execução do interpretador. A Listagem 5.31 mostra o programa principal gerado para a especificação de *LEE*.

```
1  {
2  module Syntactic where
3  import Char
4  import Lexer
5  import NotusDefault
6  import DataModule
7  import NotusFunctions
8  }
9
10 %name parse exp
11 %tokentype { Token }
12
13 %token
14         num                     { T_num $$ }
15         "+"                     { T_token0 }
16         "*"                     { T_token1 }
17         "("                     { T_token2 }
18         ")"                     { T_token3 }
19 %%
20 exp    : exp "+" term { Exp6 $1 $3 }
21        | term          {  $1 }
22 term    : term "*" factor  { Exp7 $1 $3 }
23         | factor  {  $1 }
24 factor    : num   {  $1 }
25           | "(" exp ")"  {  $2 }
26
27 {
28
29 happyError :: [Token] -> a
30 happyError _ = error "Parse error"
31
32 }
```

Listagem 5.29: Arquivo de especificação sintática *Syntactic.y* para o *Happy*

5.12 Conclusões

Neste capítulo foram apresentadas as técnicas utilizadas no compilador para a construção do módulo principal do interpretador *Haskell*, gerado a partir de definições modulares, em *Notus*, da semântica da linguagem. Assim como ocorre com a geração do reconhecedor sintático da linguagem especificada, a compilação da especificação semântica envolve a aplicação de uma série de técnicas para permitir a escrita modular dessas definições, de forma que o projetista abstraia os problemas provenientes da união dos componentes semânticos para a geração de código. É importante observar que a versão atual do compilador *Notus*, apresentada neste trabalho, não implementa as construções de escalabilidade da especificação semântica, constituindo-se em trabalho

```
1  module DataModule where
2
3  data Exp = Exp4 Int
4           | Exp6 Exp Exp
5           | Exp7 Exp Exp
6         deriving Eq
```

Listagem 5.30: Módulo `DataModule` contendo o *datatype* `Exp`

```
1  module Main where
2
3  import Lexer
4  import Syntactic
5  import DataModule
6  import NotusDefault
7  import NotusFunctions
8  import Data.Bits
9  import System
10
11 ---------- Functions ----------
12
13 dProg :: (Exp -> String)
14 dProg =  (\exp18-> ( integerToString ( dExp exp18 ) ) )
15
16 dExp :: (Exp -> Int)
17 dExp = (\exp21-> case exp21 of {
18 (Exp4 int9) ->int9 ;
19 (Exp6 exp10 exp11 ) -> ( dExp exp10 )  +  ( dExp exp11 ) ;
20 (Exp7 exp16 exp17 ) -> ( dExp exp16 )  *  ( dExp exp17 )
21 })
22
23 main = do
24   (x:xs) <- getArgs
25   progL <- readFile x
26   sWriterNotus (head(matchOutFiles [(Stdout)] xs))(dProg
   (parse (alexScanTokens progL)))
```

Listagem 5.31: Programa principal do interpretador da linguagem *LEE*

futuro a extensão do compilador para suporte dessas construções.

Os domínios sintáticos e semânticos de uma especificação em *Notus* são representados por *datatypes* em *Haskell*. Para que seja possível verificar a relação de tipos e subtipos entre os domínios, o compilador cria um grafo dirigido, cujos vértices são domínios e as arestas representam as relações de subtipagem existentes em uma definição. O grafo de relação de domínios é usado para verificar a compatibilidade entre os tipos das declarações de funções e suas respectivas definições. A etapa de geração

de código das definições também utiliza o grafo de domínios para realizar injeções e projeções nos *datatypes* correspondentes. Assim, se uma expressão do tipo t' é usada onde o seu supertipo t é esperado, a expressão é injetada no tipo t. De forma análoga, se uma expressão do tipo t' é usada onde o seu subtipo t é esperado, a expressão é projetada do tipo t.

Os domínios semânticos da especificação são analisados verificando-se a presença de possíveis ambigüidades em suas definições. Este algoritmo é executado para evitar o cenário em que um tipo t' é usado onde um tipo t é esperado e, nesse ponto, não é possível determinar a seqüência de injeções ou projeções a serem realizadas entre esses tipos, pois existe mais de um caminho, no grafo de domínios, entre esses tipos. Com a detecção de ambigüidades este cenário é reportado como um erro pelo compilador.

Adicionalmente, este capítulo apresentou como as definições de funções, escritas de forma modular em *Notus*, são ordenadas para se adequarem à forma seqüencial como *Haskell* as processa.

O compilador *Notus* é fruto de 11 meses de implementação e é composto por 18.000 linhas de código distribuídas em 153 classes e aspectos. O capítulo seguinte apresenta especificações modulares adaptadas de [Gordon, 1979] submetidas ao compilador *Notus*, com o objetivo de validar o seu funcionamento. Os interpretadores produzidos foram testados com programas em suas linguagens e para algoritmos conhecidos, como o cálculo da série de *Fibonacci* e das funções fatorial e de *Ackermann*. Os testes realizados são detalhados no próximo capítulo.

Capítulo 6

Avaliação

O compilador desenvolvido foi testado com especificações modulares em semântica denotacional para linguagens experimentais, escritas incrementalmente. Um teste inicial foi realizado com a linguagem *Nano* definida no manual da linguagem *Notus* [Tirelo e Bigonha, 2006] (veja Seção 6.1).

As linguagens *Tiny* (veja Seção 6.2) e *Small* (veja Seção 6.3), extraídas do livro do Gordon [Gordon, 1979], foram também implementadas. Os interpretadores gerados pelo compilador *Notus* para essas linguagens foram usados para executar alguns programas em suas linguagens, e os resultados esperados foram obtidos.

Durante a especificação dessas linguagens foi necessário definir uma gramática concreta, em conformidade com a gramática abstrata apresentada no livro de Gordon, já que a especificação sintática em *Notus* requer ambas. Alguns domínios semânticos também foram criados, para que fosse possível instanciar seus elementos. Além disso, foram necessárias pequenas modificações nas equações semânticas, requeridas para adequar a notação apresentada por Gordon às construções de *Notus*. Para fins de teste dos recursos de extensibilidade de *Notus* e dos algoritmos apresentados, as definições foram modularizadas de forma que se pudessem observar os problemas apresentados neste trabalho. Os interpretadores gerados foram testados visando verificar a correção das soluções implementadas pelo compilador de *Notus*.

6.1 A Linguagem *Nano*

Esta seção apresenta a sintaxe informal e a descrição dos módulos da especificação em *Notus* de uma linguagem simples contendo expressões e comandos. A especificação completa *Notus* para a linguagem *Nano* e o seu interpretador são mostrados no Apêndice B. Programas de exemplo escritos em *Nano* são mostrados na Seção 6.1.2, assim como as saídas geradas pelo seu interpretador. Trechos de código da especificação

e do interpretador de *Nano* são apresentados na Seção 6.1.3.

6.1.1 Definição da Linguagem

Um programa em *Nano* é iniciado pela leitura de um valor associado a uma variável, seguida pela execução de um comando, e finalizado com a escrita do valor de uma expressão na saída padrão. Os comandos existentes em *Nano* são os de atribuição, condicional e repetição. As expressões em *Nano* são números, booleanos, aritméticas, relacionais e lógicas. A sintaxe informal de *Nano* é dada a seguir, onde exp_1, exp_2, $\cdots$, exp_n representam expressões, comm_1, comm_2, $\cdots$, comm_n representam comandos e id_1, id_2, $\cdots$, id_n representam identificadores.

```
prog  →  "prog" id comm exp

comm  →  id ":=" exp | "if" exp "then" comm1 "else" comm2
      |  "while" exp "do" comm | "begin" comm+−";" "end"

exp   →  exp1 "or" exp2 | exp1 "and" exp2 | exp1 "<" exp2
      |  exp1 ">" exp2 | exp1 "+" exp2 | exp1 "-" exp2
      |  exp1 "*" exp2 | exp1 "/" exp2 | "not" exp
      |  "-" exp | id | num | bool
```

A especificação de *Nano* é composta pelos seguintes módulos:

- *Kernel*. Define que espaços em branco e comentários são ignorados pelo analisador léxico.
- *Expressions*. Define todos os componentes léxico, sintático e semântico de expressões, como os domínios, *tokens*, gramáticas concreta e abstrata e as equações semânticas que descrevem a denotação dessas expressões.
- *NewExpressions*. Adiciona novas expressões à especificação de *Nano*, a partir da extensão da gramática de expressões do módulo *Expressions*, bem como a definição das equações semânticas para estas novas expressões.
- *Commands*. Define todos os componentes léxico, sintático e semântico de comandos, como os domínios, *tokens*, gramáticas concreta e abstrata e as equações semânticas que descrevem a denotação desses comandos.
- *Main*. Define a sintaxe para um programa em *Nano* e a equação semântica de denotação desse programa.

É importante destacar que o módulo *NewExpressions* foi escrito como uma extensão da linguagem *Nano*, após a escrita dos demais módulos de sua definição. Os módulos em *Notus* que compõem a especificação de *Nano* são apresentados no Apêndice B.

6.1.2 Programas em Nano

Esta seção apresenta programas escritos em *Nano*, que foram executados pelo interpretador gerado pelo compilador *Notus* (veja Seção B.2 do Apêndice B). Os programas, as entradas e as saídas produzidas são mostrados a seguir.

```
read a;
begin
  b:=5;
  c:=2
end;
write (a+b)*c
```

Entradas	Saídas
5	20
10	30

Figura 6.1: Programa em *Nano testeExp.n*.

```
read a;
while a<10
  do begin
    b:=b+10;
    a:=a+1
  end;
write b
```

Entradas	Saídas
5	50
10	0

Figura 6.2: Programa em *Nano testeWhile.n*.

```
read a;
if a<10
  then b:=10
  else b:=20;
write a+b
```

Entradas	Saídas
5	15
20	40

Figura 6.3: Programa em *Nano testeIf.n*.

```
read a;
begin
  b:=1;
  while a>1
    do begin
      b:=b*a;
      a:=a-1
    end
end;
write b
```

Entradas	Saídas
5	120
7	3628800

Figura 6.4: Programa *Nano fat.n* que calcula o fatorial de um inteiro dado.

```
read a;
if a>10
  then a:=a-10
  else a:=a+(10-a);
write a + True
```

Entradas	Saídas
_	program error

Figura 6.5: Programa em *Nano testeError.n*.

6.1.3 Extratos do Código Gerado Para *Nano*

Esta seção apresenta trechos da definição de *Nano* em *Notus*, e o código correspondente em *Haskell* gerado pelo compilador.

A Listagem 6.1 mostra trechos da especificação de *Nano* referentes às definições sintática e semântica da equação de denotação para a expressão aritmética de adição. A variável de gramática `addexp` define a sintaxe da expressão *"+"* por meio da produção `addexp "+" multexp`. A função de denotação de expressão recebe como parâmetros a expressão a ser avaliada e o estado corrente de associações de variáveis e retorna o valor dessa avaliação ou erro. A equação para `exp1 "+" exp2` avalia `exp1` e `exp2` em um estado `state`, e se essas expressões resultarem em valores inteiros associados aos padrões `int1` e `int2`, o seu resultado é dado por `int1 + int2`, caso contrário, um erro é retornado.

```
1  addexp : Exp ::= addexp "+" multexp | //(...)
2
3  function dexp : Exp -> State -> Ans;
4  dexp [exp1 "+" exp2] state = let {
5    e1 = dexp exp1 state;
6    e2 = dexp exp2 state} in
7      case e1 of{
8        int1 -> case e2 of {
9                  int2 -> int1 + int2;
10                  _   -> error };
11       _   -> error};
```

Listagem 6.1: Definições sintática e semântica em *Notus* para adição em *Nano*.

O trecho da definição da gramática de *Nano* gerado pelo compilador *Notus* para o *Happy* referente à expressão *"+"* é mostrado na Listagem 6.2. A produção da gramática addexp "+" multexp retorna o nodo de *AST* Exp39 $1 $3 quando reduzida pelo analisador sintático de *Nano*. O nodo de *AST* retornado é um elemento do *datatype* `Exp` criado pelo compilador *Notus* para representar a gramática abstrata de expressões de *Nano*, onde neste exemplo, o construtor `Exp39` representa a expressão de adição e $1 e $2 armazenam, respectivamente, os valores dos não terminais `addexp` e `multexp`.

```
addexp    : addexp "+" multexp  { Exp39 $1 $3 }
```

Listagem 6.2: Gramática *Happy* para expressão aritmética de adição de *Nano*

O trecho de código *Haskell* da função `dexp` referente à equação de denotação para a expressão aritmética de adição é mostrado na Listagem 6.3. A função `dexp` espera um valor pertencente ao *datatype* `Exp`, que representa o domínio sintático `Exp`, ou seja,

`dexp` recebe elementos da AST de expressão de *Nano* representados por elementos do *datatype* `Exp`. O valor `Exp39 exp147 exp148` é um elemento do datatype *Exp* e, como mostrado na Listagem 6.2, corresponde ao nodo de *AST* da expressão *"+"*. O resultado da avaliação das expressões `exp147` e `exp148` que compõem o nodo de *AST* são armazenados nas variáveis `a0` e `a1` do tipo `Ans`. Em seguida verifica-se se `a0` e `a1` são do tipo **Int** por meio do comando **case** que injeta o tipo **Int** no tipo `Ans`, a partir dos *datatypes* `Ans` e `Value` do interpretador de *Nano* mostrados na Listagem 6.4. Caso `a0` e `a1` sejam inteiros, seus valores são associados pelas injeções realizadas à `int230` e `int231`, e o resultado da expressão é dado pela soma de `int230` e `int231`. Como o tipo de retorno de `dexp` é *Ans*, a soma (`int230 + int231`) é injetada no tipo `Ans`. Em caso de erro, a expressão resulta no valor de `Ans` correspondente a `error`.

```
1  dexp :: (Exp -> (State -> Ans))
2  dexp = ( (\exp204-> case exp204 of {
3  -- (...)
4  (Exp39 exp147 exp148 ) ->(\state226 ->let {
5    a0 = ( ( dexp exp147 ) state226 ) ;
6    a1 = ( ( dexp exp148 ) state226 ) }in  case  a0 of{
7     (U94(Value30 int230)) -> case  a1 of{
8       (U94(Value30 int231)) -> (U94 (Value30 (int230 + int231 )));
9       _ -> (E95 (Enum5__Error))
10     };
11    _ -> (E95 (Enum5__Error))});
12 -- (...)
13 }
```

Listagem 6.3: Código Haskell gerado para interpretar expressão aritmética de adição de *Nano*

```
1  data Ans = U94 Value
2           | E95 Enum5
3           deriving Eq
4
5  data Enum5 = Enum5__Error
6           deriving Eq
7
8  data Value = Value30 Int
9             | Value32 Bool
10          deriving Eq
```

Listagem 6.4: *Datatypes* `Ans` e `Value` do interpretador de *Nano*

6.2 A Linguagem *Tiny*

A linguagem *Tiny* foi especificada em *Notus* utilizando semântica denotacional com continuações. A especificação completa em *Notus* para a linguagem *Tiny* e o seu interpretador são mostrados no Apêndice C. Programas de exemplo escritos em *Tiny* são mostrados na Seção 6.2.2, assim como as saídas geradas pelo seu interpretador. Trechos de código da especificação e do interpretador de *Tiny* são apresentados na Seção 6.2.3.

6.2.1 Definição da Linguagem

A linguagem *Tiny* possui expressões e comandos. Ambos podem conter identificadores que são sequências de letras e dígitos iniciadas com uma letra, além de números que são sequências de dígitos, e os valores booleanos `"true"` e `"false"`. A sintaxe informal de *Tiny* é dada a seguir, onde e, e_1, e_2, ... representam expressões, c, c_1, c_2, ... representam comandos e id, id_1, id_2, ... representam identificadores.

c → id ":=" e | "output" e | "if" e_1 "then" e_2 "else" e_3
| "while" e "do" c | c_1 ";" c_2

e → e "=" e | e "+" e | "not" e | "read" | id | num | "true" | "false"

A especificação de *Tiny* é composta pelos seguintes módulos e pacotes:

- *Kernel*. Define que espaços em branco e comentários são ignorados pelo analisador léxico.
- *Auxiliary*. Define funções auxiliares utilizadas pelas equações de denotação dos construtos de *Tiny*.
- Módulo *Domain*. Define os domínios semânticos de *Tiny*.
- Módulo *Semantics*. Define as funções semânticas para denotação de expressões e comandos.
- Módulo *Main*. Define a função de avaliação da *AST* responsável por iniciar a interpretação do programa fonte, e funções auxiliares usadas pela função de avaliação de *AST*.
- Pacote *Expressions*. Composto pelos módulos *CoreExps* e *NewExpressions*. O módulo *CoreExps* define todos os componentes léxico, sintático e semântico das expressões de *Tiny* definidas no livro do Gordon [Gordon, 1979], como os domínios,

tokens, gramáticas concreta e abstrata e as equações semânticas que descrevem a denotação dessas expressões. O módulo *NewExpresions* adiciona novas expressões à especificação de *Tiny*, a partir da extensão da gramática de expressões do módulo *Expressions*, e define as equações semânticas para estas novas expressões.

- Pacote *Commands*. Composto pelos módulos *CoreComm* e *NewCommands*. O pacote *CoreComm* define todos os componentes léxico, sintático e semântico dos comandos de *Tiny* definidos no livro do Gordon [Gordon, 1979], como os domínios, *tokens*, gramáticas concreta e abstrata e as equações semânticas que descrevem a denotação desses comandos. O módulo *NewCommands* adiciona o comando *do while*, a partir da extensão da gramática de comandos do módulo *Commands*, bem como a definição da equação para este comando.

6.2.2 Programas em Tiny

Esta seção apresenta programas escritos em *Tiny*, que foram executados pelo interpretador gerado pelo compilador *Notus* (veja Seção C.2 do Apêndice C). Para permitir a escrita de programas interessantes e para testar o mecanismo de extensibilidade de uma especificação em *Notus*, a gramática de expressões de *Tiny* foi estendida com novas expressões, tais como `mod` e expressão entre parênteses. Os programas, as entradas e as saídas produzidas são mostrados a seguir.

```
a := read;
fib := 1;
fibsuc := 1;
a := a - 2;
while a>0 do
  n := fib + fibsuc;
  fib := fibsuc;
  fibsuc := n;
  a:= a - 1
end;
output n
```

Entradas	Saídas
5	5
16	987

Figura 6.6: Programa em *Tiny fib.t* que calcula o *n-ésimo* termo dado como entrada da série de Fibonacci.

```
n := read;
fib := 1;
fibsuc := 1;
output fib;
output fibsuc;
n := n - 2;
while n>0 do
  a := fib + fibsuc;
  output a;
  fib := fibsuc;
  fibsuc := a;
  n:= n - 1
end
```

Entradas	Saídas
5	1 1 2 3 5
18	1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

Figura 6.7: Programa em *Tiny fib2.t* que para dado um inteiro n, lista os n primeiros termos da série de Fibonacci.

```
n := read;
while n>0 do //checking if n is prime
  m := n - 1;
  while m>1 do //for all m less than n
    iterations := 0;
    if ((n mod m) = 0) // m divides n
      then m := 0  // break
      else
        m := m - 1
    end;
    iterations := iterations + 1
  end;
  if m = iterations // no one divides n
    then output n // n is prime
    else a := 0 //nop
  end;
  n := n - 1
end
```

Entradas	Saídas
15	13 11 7 5 3 2
100	97 89 83 79 73 71 67 61 59 53 47 43 41 37 31 29 23 19 17 13 11 7 5 3 2

Figura 6.8: Programa em *Tiny prime.t* que para dado um inteiro n, lista os números primos existentes entre n e 1.

6.2.3 Extratos do Código Gerado Para *Tiny*

Esta seção apresenta trechos da definição de *Tiny* em *Notus*, e o código correspondente em *Haskell* gerado pelo compilador.

A Listagem 6.5 mostra trechos da especificação de *Tiny* referentes às definições sintática e semântica da equação de denotação para o comando *while*. A variável de gramática `simpleCom` define a sintaxe do comando `while` por meio da produção *"while"* `e` *"do"* `c` *"end"*. A função de denotação de comandos recebe como parâmetros o comando a ser avaliado e o resto do programa representado pela continuação de comandos `cont`, e, após executar a semântica do comando, retorna o resto do programa, `cont`, que segue este comando. A equação para `[` *"while"* `e` *"do"* `c` *"end"*`]` chama a função de denotação de expressões `de` para a expressão `e`, cujo resultado é associado a `v`. Se `v` for um valor booleano verdadeiro, então o comando `c` é executado e o estado resultante é passado de volta para a chamada recursiva `dc` [*"while"* `e` *"do"* `c` *"end"*] `cont`. Mas, se `v` for um booleano falso o estado resultante de `de e` é enviado diretamente para `cont`. Caso `de e` não resultar em um valor booleano um erro é retornado.

```
public simpleCom : Com ::= "while" e "do" c "end" | //(...);

function dc : Com -> Cont -> Cont;
dc ["while" e "do" c "end"] cont =
de e (\v ->
  case v of {bool ->
    if bool
      then evaluate {dc c ;dc ["while" e "do" c "end"] cont}
      else cont;
           _    -> err});
```

Listagem 6.5: Definições sintática e semântica em *Notus* para o comando *while* de *Tiny*.

O trecho de código da gramática de *Tiny* gerado pelo compilador Notus para o *Happy*, referente ao comando *while*, é mostrado na Listagem 6.6. A produção da gramática simpleCom : "while" e "do" c "end" retorna o nodo de *AST* Com44 $2 $4 quando reduzida pelo analisador sintático de *Tiny*. O nodo de AST retornado é um elemento do *datatype* `Com` criado pelo compilador *Notus* para representar a gramática abstrata de comandos de *Tiny*.

```
simpleCom     : "while" e "do" c "end"          { Com44 24 }
```

Listagem 6.6: Gramática *Happy* para o comando *while* de *Tiny*.

O trecho de código *Haskell* do interpretador de *Tiny*, referente à equação de denotação do comando *while*, é mostrado na Listagem 6.7. A função `dc` recebe como

primeiro parâmetro um valor do tipo do *datatype* Com associado a com340, que representam os comandos de *Tiny*. A equação de denotação de comandos utiliza a expressão *case* sobre o comando com340, para seleção do tipo do argumento comando. Como mostrado na Listagem 6.6 o construtor Com44 representa o valor do *datatype* Com para o comando *while*, e exp167 e com168 representam, respectivamente, a expressão e o comando do corpo do *while*. O segundo parâmetro de dc é a continuação do programa analisado, associado à variável cont364. Em seguida, a função de denotação de expressões é executada para a expressão exp167 do comando *while*. A função de denotação de expressões recebe, além da expressão, uma continuação de expressões do tipo Econt. A continuação de expressões passada para de executa a semântica do *while* de acordo com o valor resultante da avaliação de exp167. A Listagem 6.8 mostra os *datatypes* do interpretador de *Tiny* que representam continuação de expressões de tipo Econt, e valores de tipo Value, utilizados pela equação do comando *while*.

```
dc :: (Com -> (Cont -> Cont))
dc = ( (\com340-> case com340 of {
--(...)

(Com44 exp167 com168) ->
  (\cont364 ->( ( de exp167 ) (F80 (\a0 -> case  a0 of{
    ((Value29 bool367)) ->
      if ( bool367 )
        then ((dc com168) ((dc (Com44 exp167 com168  )) cont364))
        else (cont364) ;
    _  -> err })))));

--(...)
```

Listagem 6.7: Código *Haskell* gerado para interpretação do comando *while* de *Tiny*.

```
data Econt = F80  (Value -> Cont)

data Value = Value27 Int
           | Value29 Bool
        deriving Eq
```

Listagem 6.8: *Datatypes*Econt e Value do interpretador de *Tiny*.

6.3 A Linguagem *Small*

A linguagem *Small* foi especificada em *Notus* utilizando semântica denotacional padrão. A especificação completa em *Notus* para a linguagem *Small* e o seu interpretador são mostrados no Apêndice D. Programas de exemplo escritos em *Small* são mostrados na Seção 6.3.2, assim como as saídas geradas pelo seu interpretador. Trechos de código da especificação e do interpretador de *Small* são apresentados na Seção 6.3.3.

6.3.1 Definição da Linguagem

Programas na linguagem *Small* podem ser compostos por expressões, comandos e declarações de funções, procedimentos, variáveis e constantes, iniciados sempre com a palavra-chave `"program"`. A especificação de *Small* é composta pelos domínios sintáticos primitivos dos identificadores, que são seqüências de letras e dígitos iniciadas com uma letra, das contantes básicas representadas por números, e pelo conjunto de operadores binários $\{+, -, *, /, <, >, ==\}$, além dos domínios das variáveis de gramática. A sintaxe informal de *Small* é dada a seguir, onde e, e_1, e_2, ... representam expressões, c, c_1, c_2, ... representam comandos, d, d_1, d_2, ... representam declarações, id, id_1, id_2, ... representam identificadores e bas é uma constante básica.

$$
\begin{array}{lcl}
\mathsf{p} & \rightarrow & \texttt{"program"}\ \mathsf{c} \\
\\
\mathsf{e} & \rightarrow & \mathsf{bas} \mid \texttt{"true"} \mid \texttt{"false"} \mid \texttt{"read"} \mid \mathsf{id} \mid \mathsf{e}_1\ \texttt{"("}\ \mathsf{e}_2\ \texttt{")"} \\
 & \mid & \texttt{"if"}\ \mathsf{e}_1\ \texttt{"then"}\ \mathsf{e}_2\ \texttt{"else"}\ \mathsf{e}_3 \mid \mathsf{e}_1\ \mathsf{opr}\ \mathsf{e}_2 \\
\\
\mathsf{c} & \rightarrow & \mathsf{e}_1\ \texttt{":="}\ \mathsf{e}_2 \mid \texttt{"output"}\ \mathsf{e} \mid \texttt{"call"}\ \mathsf{e}_1\ \texttt{"("}\ \mathsf{e}_2\ \texttt{")"} \\
 & \mid & \texttt{"if"}\ \mathsf{e}\ \texttt{"then"}\ \mathsf{c}_1\ \texttt{"else"}\ \mathsf{c}_2 \mid \texttt{"while"}\ \mathsf{e}\ \texttt{"do"}\ \mathsf{c} \\
 & \mid & \mathsf{c}_1\ \texttt{";"}\ \mathsf{c}_2 \mid \texttt{"begin"}\ \mathsf{d}\ \texttt{";"}\ \mathsf{c}\ \texttt{"end"} \\
\\
\mathsf{d} & \rightarrow & \texttt{"const"}\ \mathsf{id}\ \texttt{"="}\ \mathsf{e} \mid \texttt{"var"}\ \mathsf{id}\ \texttt{"="}\ \mathsf{e} \mid \texttt{"proc"}\ \mathsf{id}\ \texttt{"("}\ \mathsf{i}_1\ \texttt{")"}\ \mathsf{c} \\
 & \mid & \texttt{"fun"}\ \mathsf{id}\ \texttt{"("}\ \mathsf{i}_1\ \texttt{")"}\ \mathsf{e} \mid \mathsf{d}_1\ \texttt{";"}\ \mathsf{d}_2 \\
\\
\mathsf{opr} & \rightarrow & \texttt{"+"} \mid \texttt{"-"} \mid \texttt{"*"} \mid \texttt{"/"} \mid \texttt{">"} \mid \texttt{"<"} \mid \texttt{"=="}
\end{array}
$$

A especificação de *Small* é composta pelos seguintes módulos:

- *Kernel*. Define que espaços em branco e comentários são ignorados pelo analisador léxico.
- *Commands*. Define a gramática e as equações de denotação para os comandos de *Small* descritos no livro do Gordon [Gordon, 1979].

- *Declarations*. Define a gramática e as equações de denotação para as declarações de *Small* descritas no livro do Gordon [Gordon, 1979].

- *Domains*. Define os domínios sintáticos e semânticos da especificação de *Small*. Os domínios desse módulo definem os domínios da semântica denotacional padrão de *Small*.

- *Expressions*. Define a gramática e as equações de denotação para as expressões de *Small* descritas no livro do Gordon [Gordon, 1979].

- *Operators*. Define a gramática e as equações de denotação para os operadores de *Small*.

- *Semantics*. Contém a declaração das funções semânticas de *Small* e a definição da gramática e da equação de denotação de um programa em *Small*.

- *Tokens*. Contém a definição dos *tokens* `id`, `bas` e `bool` de *Small*.

- *Util*. Define funções auxiliares utilizadas pelas equações de denotação dos construtos de *Small*.

- *Main*. Define a função de avaliação da *AST* responsável por iniciar a interpretação do programa fonte. Funções auxiliares para manipulação de seqüências de entrada e saída de programas em *Small* também estão definidas nesse módulo. *Main* é o módulo principal da especificação de *Small*.

6.3.2 Programas em Small

Esta seção apresenta programas escritos em *Small*, que foram executados pelo interpretador gerado pelo compilador *Notus* (veja Seção D.2 do Apêndice D). Os programas, as entradas e as saídas produzidas são mostrados a seguir.

```
program
  begin
    var d = read;
    if d == 1
      then
        begin
          const c = 10;
          c := 20;
          output c
        end
      else
        begin
          var c = 10;
          c := 20;
          output c
        end
    end
  end
```

Entradas	Saídas
1	program error
2	20

Figura 6.9: Programa em *Small constTest.small* que testa atribuição indevida a uma constante.

```
program
  begin
    fun fat(n)
      if n == 0
        then 1
        else n * fat(n-1)
      end
    end;
    var x = read;
    output fat(x)
  end
```

Entradas	Saídas
5	120
10	3628800
12	479001600

Figura 6.10: Programa em *Small fat.small* que calcula recursivamente o fatorial de um inteiro dado.

```
program
  begin
    proc fib(n)
      begin
        var f1 = 1;
        var f2 = 1;
        var i = 2;
        output f1;
        output f2;
        while i < n
          do
            begin
              var t = f2;
              f2 := f1 + f2;
              f1 := t;
              output f2;
              i := i + 1
            end
          end
      end
    end;
    var n = read;
    call fib(n)
  end
```

Entradas	Saídas
5	1 1 2 3 5
20	1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765

Figura 6.11: Programa em *Small fib-list.small* que para dado um inteiro n, lista os n primeiros termos da série de Fibonacci.

6.3.3 Extratos do Código Gerado Para *Small*

Esta seção apresenta trechos da definição de *Small* em *Notus*, e o código correspondente em *Haskell* gerado pelo compilador.

A Listagem 6.9 mostra as definições da sintaxe e das equações semânticas de denotação para a declaração e a chamada de um procedimento de *Small*. Uma declaração de procedimento é realizada por meio da palavra-chave *"proc"* seguida pelo nome, parâmetro e o corpo do método. A chamada de um procedimento é feita por meio da palavra reservada *"call"*, seguida pelo nome e argumento do procedimento.

A equação de denotação para a declaração de um procedimento associa o valor do procedimento ao seu identificador em um pequeno *environment* passado para o resto do programa, representado pela continuação `dc`. O valor do procedimento, a ser executado quando o procedimento for chamado, avalia o corpo do procedimento no *environment* da declaração do procedimento, acrescido da associação do parâmetro real ao parâmetro formal. Adicionalmente, o identificador do procedimento é associado a si mesmo em seu *environment*, para permitir chamadas recursivas. Para permitir que variáveis sejam passadas como argumentos é necessário criar uma posição no *store*, usando-se a função auxiliar `ref` (veja Seção D.8), na associação do argumento ao parâmetro formal.

A chamada de um procedimento é representada pelo nodo de *AST* [*"procCall"* id e] onde `id` e `e` são, respectivamente, o nome e o argumento do procedimento. A equação de denotação de chamada de procedimento avalia `id` e seu valor é verificado para assegurar que se trata de um procedimento `proc`. Em seguida, o valor do argumento `e` é avaliado e seu valor é passado para `proc`, e finalmente o `store` resultante do procedimento `proc` é passado para o resto do programa representado pela continuação `cc`.

A Listagem 6.10 mostra o trecho da gramática *Happy* correspondente à declaração e a chamada de um procedimento em *Small*. O nodo de *AST*, referente à declaração de um procedimento Dec75 $2 $4 $6, é um valor do *datatype* `Dec`, criado pelo compilador *Notus* para representar o domínio sintático das declarações de *Small*. De forma análoga, o nodo de *AST* Com68 $2 $4, que representa o comando de chamada a um procedimento, é um valor do *datatype* `Com`, que representa o domínio sintático dos comandos de *Small*.

```
simpleDec    : "proc" id "(" id ")" c "end"     { Dec75 $2 $4 $6 }

simpleCom    : "call" id "(" e ")"              { Com68 $2 $4 }
```

Listagem 6.10: Gramática *Happy* para declaração e chamada de procedimento de *Small*.

A Listagem 6.11 mostra o trecho do código do interpretador de *Small* referente à equação de denotação de declaração e chamada de um procedimento em *Small*. A

```
1  module Declarations
2
3    simpleDec : Dec ::= "proc" id "(" id1 ")" c "end"
4                      | // (...);
5
6    function ddec  : Dec -> Env -> Dc -> Cc;
7    ddec ["proc" id "(" id1 ")" c "end"] env dc =
8      let {proc = (\cc ev ->
9        ref (\loc ->
10         dcom c env[id1 <- loc,id <- proc] cc) ev)}
11     in dc newEnv[id <- proc];
12
13   //(...)
14 end
15
16 module Commands
17
18   simpleCom : Com ::= "call" id "(" e ")": ["procCall" id e]
19                     | // (...)
20
21   function dcom  : Com -> Env -> Cc -> Cc;
22   dcom ["procCall" id e] env cc =
23     evaluate {dexp [id] env;
24               isProc (\proc -> evaluate {dr e env; proc; cc})
25              };
26   //(...)
27 end
```

Listagem 6.9: Definições sintática e semântica em *Notus* para a declaração e comando chamada de um procedimento de *Small*.

função semântica do interpretador para avaliação de declarações de *Small* recebe como primeiro parâmetro o nodo de *AST* correspondente à declaração. No caso da declaração de um procedimento, como mostrado na Listagem 6.10, o nodo de *AST* é Dec75 id255 id256 com257, onde id255 é o nome do procedimento, id256 é o parâmetro e com257 é o corpo do procedimento. Adicionalmente, a função ddec recebe um *environment* env534 e a continuação de declaração dc530. A avaliação do corpo do procedimento procedimento é representada pela variável proc532, que é associada ao nome do procedimento id255 no *environment* env528 da declaração nas linhas 10 e 11. O parâmetro id256 é associado à nova posição de *store* loc535 retornada pela função auxiliar ref na linha 6. A continuação do resto do programa dc, representada por um *datatype* funcional cujo valor é F116 (Env −> Cc) é projetada na linha 14 para receber um pequeno *environment*, retornado pela função newEnv, acrescido da associação do nome do procedimento à variável proc532.

```
1  ddec :: (Dec -> (Env -> (Dc -> Cc)))
2  ddec = ( (\dec507-> case dec507 of {
3  --(...)
4  (Dec75 id255 id256 com257 ) ->(\env528 ->(\dc530 ->(let {
5    proc532 = (F117 (\cc533 -> (F115 (\ev534 ->
6      ((case (ref (F115 (\(D108(U102 loc535)) ->
7        (((dcom com257 )
8          (F112 (update
9            (case env528 of (F112 funDomains539) -> funDomains539)
10             [id256,id255]
11             [(U123 (U102 loc535)),(U123 (U104 proc532))])))cc533))))
12           of (F115 funDomains540) -> funDomains540) ev534)))))}
13   in ((case dc530 of
14     ((F116 funDomains541)) -> funDomains541)
15       (F112 (update
16         (case newEnv of (F112 funDomains542) -> funDomains542)
17         [ id255 ]
18         [(U123 (U104 proc532))]))
19     ))));
20
21 dcom :: (Com -> (Env -> (Cc -> Cc)))
22 dcom = ( (\com435-> case com435 of {
23 --(...)
24 (Com68 id218 exp219) ->(\env459 ->(\cc461 ->
25   (((dexp (Exp61 id218))env459)
26   (isProc (F115 (\ (D108(U104 proc463)) ->
27     (((dr exp219) env459)
28     ((case proc463 of ((F117 funDomains467)) -> funDomains467)
29     cc461))))))));
```

Listagem 6.11: Código *Haskell* gerado para interpretação de declaração e chamada de procedimento de *Small*.

A função semântica do interpretador para avaliação de comandos de *Small* recebe como primeiro parâmetro o nodo de *AST* correspondente ao comando, um *environment* e a continuação do resto do programa. O comando de chamada de procedimento é representado pelo nodo de AST Com68 id218 exp219, onde id218 é o nome do procedimento e exp219 é o argumento. O identificador do procedimento id218 é avaliado pela função dexp na linha 25, obtendo-se o procedimento proc463. O argumento de chamada exp219 é avaliado pela função dr, em uma continuação que executa o corpo do procedimento proc463, na continuação da chamada do procedimento cc461.

6.4 Análise de Desempenho do Código Gerado

Objetivando analisar empiricamente o desempenho do código *Haskell* gerado pelo compilador *Notus*, experimentos foram realizados diretamente sobre o código gerado e indiretamente sobre os interpretadores gerados para as linguagens definidas de exemplo.

Os experimentos realizados diretamente sobre o código gerado consistem em definir em *Notus* funções conhecidas. Estes experimentos visam medir os possíveis custos de *overhead* gerados pela etapa de tradução da especificação semântica de uma linguagem em *Notus* para *Haskell*. O código *Haskell* gerado pelo compilador para as funções mencionadas é então executado e os tempos de execução são medidos e comparados aos tempos para funções equivalentes escritas diretamente em *Haskell*.

Os experimentos realizados de forma indireta consistem em medir o tempo de execução dos interpretadores gerados para programas na linguagem definida. As seções seguintes apresentam os testes de desempenho realizados e os resultados obtidos.

6.4.1 Análise do *Overhead* da Tradução

Expressões em *Notus* foram baseadas na linguagem *Haskell*. Por isso é possível escrever funções equivalentes nessas linguagens desde que se usem apenas tipos primitivos. Dessa forma para que fosse possível calcular o *overhead* da tradução de equações de *Notus* para funções em *Haskell*, implementações das funções recursivas para cálculo do fatorial, da série de Fibonacci e da função de Ackermann foram escritas em ambas linguagens.

O módulo *Notus* contendo apenas as definições para as funções de teste é mostrado na Listagem 6.12. Note que não foram definidos componentes de especificação léxica e sintática de linguagens, já que o objetivo deste teste é analisar o código *Haskell* gerado para funções *Notus*. Além das funções de teste, foram criadas duas funções auxiliares, `fatN` e `fibonacciN`. Essas funções foram usadas para possibilitar múltiplas chamadas às funções fatorial e de *Fibonacci*.

A Listagem 6.13 mostra o módulo *Haskell* gerado pelo compilador a partir da definição da Listagem 6.12. A não especificação, no módulo *Notus*, de uma gramática, e conseqüentemente de seu símbolo inicial, implica que o código *Haskell* gerado não contém a função *main* que inicia a execução do programa. Dessa maneira, o módulo gerado foi editado a cada teste realizado, incluindo-se uma função *main* que invocasse, a cada momento, uma das funções implementadas.

```
module Main

function fat : Int -> Int;
fat 0 = 1;
fat n = n * fat (n - 1);

function fatN : Int -> Int -> Int -> Int;
fatN 0 n r = r;
fatN v n r = fatN (v-1) n (fat n);

function ackermann : Int -> Int -> Int;
ackermann 0 n = n + 1;
ackermann m 0 = if (m > 0)
                  then ackermann (m - 1) 1
                  else 0;
ackermann m n = if (m > 0) and (n > 0)
                  then ackermann (m - 1) (ackermann m (n - 1))
                  else 0;

function fibonacci : Int -> Int;
fibonacci 0 = 0;
fibonacci 1 = 1;
fibonacci n = fibonacci (n-1) + fibonacci (n-2);

function fibonacciN : Int -> Int -> Int -> Int;
fibonacciN 0 m f = f;
fibonacciN v m f = fibonacciN (v - 1) m (fibonacci m);

end
```

Listagem 6.12: Módulo *Notus* contendo as funções analisadas.

```
module Main where

fat :: (Int -> Int)
fat =
  (\int11-> case int11 of {
    (0) ->1 ;
    (_) ->( ( int11 ) * ( fat( int11 - 1 ) ))
  })

fatN :: (Int -> (Int -> (Int -> Int)))
fatN =
  (\int13-> case int13 of {
    (0) ->(\int14 ->(\int16 ->int16 ));
    (_) ->(\int18 ->(\int20 ->
      ( ( ( fatN ((int13 - 1 ) ) ) int18 ) ( ( fat int18 )))))
})

ackermann :: (Int -> (Int -> Int))
ackermann =
  (\int24-> case int24 of {
    (0) ->( (\int25 ->( ( int25 ) + ( 1 ) ) ));
    (_) -> (\int27-> case int27 of {
      (0) ->if ( int24 > 0 )
              then (( ackermann ( int24 - 1 ) ) 1 )
              else ( 0 ) ;
      (_) ->if  ( ( int24 > 0 ) && ( int27  >  0 ) )
              then ( ackermann ( int24 - 1 )
                               ( ackermann int24 )
                               ( ( int27 - 1 ))
                   )
              else ( 0 )
})
})

fibonacci :: (Int -> Int)
fibonacci =
  ( (\int31-> case int31 of {
    (0) ->0 ;
    (1) ->1 ;
    (_) ->( fibonacci ( int31 - 1 )) + ( fibonacci (int31 - 2))
}))

fibonacciN :: (Int -> (Int -> (Int -> Int)))
fibonacciN =
  (\int39-> case int39 of {
    (0) ->(\int40 ->( (\int42 ->int42 )));
    (_) ->(\int44 ->
      (\int46 ->((( fibonacciN ( int39 - 1 ) ) int44 )
                 (fibonacci int44 ))
       ))
})
```

Listagem 6.13: Funções em *Notus*.

A Listagem 6.14 mostra o código para as funções de teste escrito diretamente em *Haskell*.

```
module Main where

fat :: Int -> Int
fat 0 = 1
fat n = n * fat (n - 1)

fatN :: Int -> Int -> Int -> Int
fatN 0 n r = r
fatN v n r = fatN (v-1) n (fat n)

fibonacci :: Int -> Int
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci (n-1) + fibonacci (n-2)

fibonacciN :: Int -> Int -> Int -> Int
fibonacciN 0 m f = f
fibonacciN v m f = fibonacciN (v - 1) m (fibonacci m)

ackermann :: Int -> Int -> Int
ackermann 0 n = (n + 1)
ackermann m 0 = if (m > 0) then ackermann (m - 1) 1 else 0
ackermann m n = if (m > 0) && (n > 0)
                  then ackermann (m - 1) (ackermann m (n - 1))
                  else 0
```

Listagem 6.14: Funções escritas diretamente em *Haskell*.

O código *Haskell* gerado pelo compilador *Notus* e o escrito diretamente foram submetidos ao compilador *GHC*, e os programas gerados foram executados para as três funções de teste com entradas diferentes. Os tempos de execução foram coletados por meio do comando *time* do UNIX. As medidas de tempo de execução apresentadas nesta seção foram obtidas a partir da execução dos programas 33 vezes, o que, segundo [Triola, 1998], garante que as médias amostrais obtidas são próximas da média populacional.

Os tempos médios de execução para a função fatorial são mostrados na Tabela 6.1. Essa função foi chamada um milhão de vezes para cada uma das entradas, com o objetivo de aumentar a precisão dos tempos medidos. A maior entrada possível, considerando que um inteiro em *Haskell* possui 32 bits, foi 12.

Os tempos médios de execução para a função de cálculo do *n-ésimo* termo da série de Fibonacci são mostrados na Tabela 6.2. Essa função também foi chamada um milhão de vezes para cada uma das entradas, para aumentar a precisão dos tempos medidos.

Entradas	Haskell Gerado (s)	Haskell (s)
5	0,0406	0,0414
10	0,0466	0,0448
12	0,0479	0,0452

Tabela 6.1: Tempo de execução para a função fatorial.

Entradas	Haskell Gerado (s)	Haskell (s)
10	0,049	0,042
15	0,049	0,043
20	0,051	0,043
25	0,066	0,056
30	0,253	0,208
35	2,279	1,898

Tabela 6.2: Tempo de execução para a função Fibonacci.

Os tempos médios de execução para a função de *Ackermann* são mostrados na Tabela 6.3.

Entradas	Haskell Gerado (s)	Haskell (s)
(3,4)	0,007	0,009
(3,5)	0,015	0,016
(3,6)	0,038	0,040
(3,7)	0,128	0,138
(3,8)	0,495	0,542
(3,9)	2,022	2,168

Tabela 6.3: Tempo de execução para a função de Ackermann.

Os tempos medidos correspondem aos tempos totais de processamento envolvendo fatores externos, tais como escalonamento e alocação de memória. Apesar disso as diferenças não passam de poucos milissegundos. Os tempos de execução para o código *Haskell* gerado pelo compilador *Notus* e o escrito diretamente em *Haskell* não apresentam grandes discrepâncias. É possível verificar, a partir de uma inspeção no código *Notus* (veja Listagem 6.12) e o código *Haskell* (veja Listagem 6.14), que as funções de teste apresentam poucas diferenças: a não ser pela palavra reservada `function` e os separadores utilizados em *Notus*, estes códigos são iguais. Mas, comparando-se o código *Haskell* gerado e o código *Haskell* escrito diretamente percebe-se que a diferença que existe é que as equações das funções geradas pelo compilador utiliza uma expressão **case**, em que os argumentos são capturados um a um por abstrações lambdas. No entanto, essa diferença não influencia os tempos de execução pois funções em *Haskell* são tratadas sempre como funções de ordem mais alta que processam um argumento

por vez e retornam uma função que recebe os demais argumentos. Dessa forma, não há impacto no desempenho se os argumentos são apresentados no lado esquerdo ou como abstrações lambda no lado direito de definições de funções. Assim, as diferenças de tempos de execução apresentadas nos experimentos realizados são atribuídas a fatores externos, e portanto, a tradução de funções fundamentais de *Notus* para *Haskell* não gera *overhead*.

6.4.2 Análise dos Interpretadores Gerados

Seções 6.1.2, 6.2.2 e 6.3.2 apresentam programas escritos nas linguagens *Nano*, *Tiny* e *Small*, executados nos interpretadores dessas linguagens gerados pelo compilador *Notus*. Esta seção objetiva analisar o desempenho desses interpretadores. Para permitir uma análise detalhada usou-se o sistema de *profiling* do compilador *GHC* [The GHC Team, 2006, Capítulo 5]. Esse sistema associa custos de execução a centros de custos. O custo refere-se ao tempo e espaço de memória necessários durante a avaliação de uma expressão. Centro de custos são anotações no programa realizadas em torno das expressões. Além do tempo total e da quantidade total de memória alocada durante a execução do programa, apresentam-se as funções com maior custo. A seguir são apresentados os programas de testes e as informações de custo obtidas na execução de programas dos interpretadores gerados, bem como uma análise sobre esses custos.

6.4.2.1 Interpretador de *Nano*

A análise de desempenho no interpretador de *Nano* foi realizada utilizando-se o programa para cálculo do fatorial de um número inteiro apresentado na Figura 6.4. No entanto, esse programa foi alterado para possibilitar que a rotina do cálculo do fatorial fosse executada 1000 vezes objetivando aumentar a precisão dos custos medidos. O programa fatorial alterado, denominado *fatN*, é apresentado na Listagem 6.15.

```
read a;
begin
  b:=1;
  times:=1000;
  while times>0
    do begin
      while a>1
        do begin
          b:=b*a;
          a:=a-1
        end;
```

```
        times := times - 1
      end
end;
write b
```

Listagem 6.15: Programa *Nano fatN.n* para cálculo do fatorial de um inteiro dado

O interpretador de *Nano* foi executado 33 vezes para o programa da Listagem 6.15, para cálculo do fatorial de 10. A média das informações de *profiling* obtidas, porcentagem do tempo total e porcentagem de alocação de memória para as funções mais caras, são apresentadas na Tabela 6.4. As funções `dexp` e `dcom` (veja Seção B.1) correspondem, respectivamente, às equações de denotação das expressões e dos comandos de *Nano*. A função `update` é a função pré-definida do compilador *Notus*, responsável pela atualização de funções, e é utilizada sempre que uma expressão de atualização aparece em uma equação (veja Seção 3.7.5.8). Na definição de *Nano* a expressão de atualização de função é utilizada nas equações de denotação do comando de atribuição e de denotação do programa. Em ambos os casos esta função é utilizada para atualização de variável no estado corrente da execução de um programa em *Nano*. A média do tempo total para o fatorial de 10, executado 1000 vezes, foi $0,05$ segundos e a média da memória total alocada, excluindo o *overhead* de *profiling*, foi de $10,62$ *MB*.

Funções	% Tempo Total	%Alocação de memória
`update`	100	92,2
`dexp`	0	4
`dcomm`	0	1,7

Tabela 6.4: Informações de *profiling* para fatorial de 10.

O esforço de processamento despendido em programas de linguagens imperativas, como *Nano*, se concentra principalmente em operações de atualização de variável. O comando mais freqüente no programa para cálculo do fatorial apresentado é o de atribuição de variável. Por isso, como apresentado na Tabela 6.4, o processamento do programa de teste é predominantemente gasto pela função `update`, de forma que o tempo gasto nas demais funções não foi significativo o suficiente para ser medido pela aplicação de *profiling*, e o consumo de memória para as demais funções soma menos de 8% da memória total usada.

6.4.2.2 Interpretador de *Tiny*

A análise de desempenho no interpretador de *Tiny* foi realizada utilizando-se o programa em *Tiny prime.t*, apresentado na Figura 6.8. O programa *prime.t* recebe como

entrada um inteiro n e lista os números primos existentes entre n e 1. O interpretador de *Tiny* foi executado 33 vezes para o programa *prime.t* para o número 500, gerando como saída a seqüência de números primos entre 500 e 1. A média das informações de *profiling* obtidas, porcentagem do tempo total e porcentagem de alocação de memória para as funções mais caras, são apresentadas na Tabela 6.5. As funções `de` e `dc` (veja Seção C.1) correspondem, respectivamente, às equações de denotação das expressões e dos comandos de *Tiny*. A função `update` é a função pré-definida do compilador *Notus*, responsável pela atualização de funções, e é utilizada sempre que uma expressão de atualização aparece em uma equação (veja Seção 3.7.5.8). Na definição de *Tiny*, a expressão de atualização de função é utilizada na equação de denotação do comando de atribuição. Esta função é utilizada para atualização de variável na memória do estado corrente da execução de um programa em *Tiny*. A média do tempo total para o cálculo dos números primos entre 500 e 1 foi $5,99$ segundos e a média da memória total alocada, excluindo o *overhead* de *profiling*, foi de $786,86$ *MB*. Note que a memória total alocada não se refere à memória necessária para execução do programa a cada momento, mas sim quanto o programa consumiu de memória durante toda a sua execução.

Funções	% Tempo Total	%Alocação de memória
`update`	90,61	80,4
`de`	6,24	11,9
`dc`	2,43	7,2

Tabela 6.5: Informações de *profiling* para cálculo dos números primos entre 1 e 500.

Assim como ocorreu com o teste do interpretador de *Nano*, o programa de teste para o interpretador de *Tiny* apresentou maior consumo de tempo e memória pela função de atualização de memória, `update`. No entanto, o programa *prime.t* apresenta, além de comandos de atribuição, os comandos condicional e de saída e expressões, como a de resto de divisão, subtração e adição, realizados iterativamente. Por esse motivo, as equações de denotação de comandos e expressões totalizaram quase 10% do tempo total e quase 20% da memória total utilizada.

6.4.2.3 Interpretador de *Small*

A análise de desempenho no interpretador de *Small* foi realizada utilizando-se o programa em *Small fib.small* da Listagem 6.16. O programa *fib.small* recebe como entrada um inteiro n e calcula o *n-ésimo* termo da série de Fibonacci recursivamente, por meio da função `fib`.

O interpretador de *Small* foi executado 33 vezes para o programa *fib.small*, para o número 11, gerando como saída o décimo-primeiro termo da série de Fibonacci, 89. A

```
program
  begin
    fun fib(n)
      if n == 1
        then 1
        else if n == 2
             then 1
             else fib(n-1) + fib(n-2)
            end
      end
    end;
    var n = read;
    output fib(n)
  end
```

Listagem 6.16: Programa *Small fib.small* para cálculo do $n-ésimo$ termo da série de Fibonacci.

média das informações de *profiling* obtidas, porcentagem do tempo total e porcentagem de alocação de memória para as funções mais caras, são apresentadas na Tabela 6.6. A função `findNew` (veja Seção D.1) procura recursivamente, em um *store* passado como argumento, a primeira posição não alocada, retornando-a. No interpretador de *Small*, essa função é chamada pela função `ref`, que é utilizada pelas equações de denotação de declarações de variáveis, funções e procedimentos, para obtenção de uma nova posição no *store* a ser associada a essas declarações, como mostrado no código de denotação para a declaração de procedimento na Seção 6.3.3. Na definição de *Small*, a expressão de atualização de funções, e portanto a função pré-definida do compilador `update`, é utilizada nas equações de denotação das declarações de contantes, variáveis, procedimentos e funções, na denotação da expressão `read`, na denotação de um programa, e na equação de denotação do comando de atribuição. A média do tempo total para o cálculo do décimo primeiro termo da série de Fibonacci foi $4,51$ segundos e a média da memória total alocada, excluindo o *overhead* de *profiling*, foi de $833,90$ *MB*.

Funções	% Tempo Total	%Alocação de memória
`update`	98,27	96,9
`findNew`	1,73	3

Tabela 6.6: Informações de *profiling* para cálculo do décimo primeiro termo da série de Fibonacci.

Assim como ocorreu nos testes dos interpretadores de *Nano* e *Tiny*, o programa de teste para o interpretador de *Small* apresentou maior consumo de tempo e memória pela função de atualização de memória, `update`. O programa *fib.small* atualiza o *environment*, a cada chamada recursiva da função `fib`, para associação dos identificadores

do parâmetro e da função. Pelo fato da execução desse programa ser marcada essencialmente pelas chamadas recursivas à função `fib`, a maior parte do tempo e da alocação de memória do programa é gasta pela função `update`. A função *findNew* também é invocada a cada chamada recursiva, porém essa função individualmente não é tão cara quanto a função `update`, representado, assim, menos de 2% do tempo total e 3% da memória total alocada.

6.5 Conclusões

Este capítulo descreve os testes realizados no compilador *Notus*. O objetivo desses testes é mostrar o poder de modularidade e extensibilidade da linguagem *Notus* e de seu compilador. O conjunto de construções presentes nas linguagens de teste definidas mostra que o uso do compilador *Notus* para especificações de linguagens reais é viável. Os interpretadores gerados para as linguagens especificadas executaram programas nessas linguagens e apresentaram tempos de execução aceitáveis.

A Seção 6.4.2 apresentou os custos de execução de programas nas linguagens *Nano*, *Tiny* e *Small* pelos interpretadores gerados para essas linguagens. Pode-se perceber que a maioria do consumo de tempo e memória foi gasta pela função de atualização `update`. De fato, a implementação dessa função pré-definida do compilador *Notus*, apresentada na Listagem 5.26 da Página 148, é cara, pois envolve busca seqüencial em lista. O alto custo de sua implementação é agravado pelo fato de a atualização do valor de variáveis ser a operação mais comum em linguagens imperativas.

Além das especificações em semântica denotacional realizadas para teste do compilador, o código *Haskell* gerado foi também avaliado comparando-se, sempre que possível, o código gerado com um código escrito diretamente em *Haskell*. Os possíveis custos de *overhead* gerados pela etapa de tradução foram avaliados e verificou-se que, para funções com domínios primitivos, não há diferença significativa no tempo de execução entre o código *Haskell* gerado pelo compilador *Notus* e um escrito diretamente em *Haskell*.

O compilador *Notus*, ao possibilitar a execução de especificações denotacionais, permite a identificação de problemas ou situações não previstas inicialmente. Por exemplo, suponha que a especificação de *Small* do livro do Gordon [Gordon, 1979] suportasse funções e procedimentos recursivos e a utilização de variáveis como parâmetros, embora essa não seja a intenção do Gordon. Se essas características fossem consideradas, a especificação apresentaria os seguintes problemas:

- na equação de denotação de declaração de funções e procedimentos, o próprio subprograma deve ser associado ao seu *environment* para permitir chamadas re-

cursivas. Caso essa associação não seja realizada, o identificador do subprograma não existirá dentro de seu corpo, gerando um erro ao se buscar esse identificador na equação de denotação de chamada;

- na equação de denotação de declaração de funções e procedimentos, é necessário criar uma posição no *store* (usando a função auxiliar `ref` da Listagem D.8), para associar o argumento ao parâmetro formal, permitindo que variáveis passadas como argumentos para funções e procedimentos sejam atualizadas em seu corpo. Caso o referenciamento não seja feito, ao se atualizar uma variável passada como argumento de uma função/procedimento, a posição do *store* retornada pela avaliação da variável no comando de atribuição não é visível no procedimento e um erro é encontrado ao se tentar atribuir o novo valor a essa posição;

- em chamadas de procedimentos e funções, o argumento deve ser avaliado como uma expressão do lado direito (utilizando-se a equação `dr` da Listagem D.5), para que seu valor seja recuperado. Originalmente, o argumento é avaliado com a equação `dexp`, que resulta em uma posição não disponível no contexto do corpo do procedimento, para o caso em que o argumento é uma variável.

Capítulo 7

Conclusões

Neste texto, apresentou-se a implementação de um compilador para a linguagem *Notus*, que traduz descrições formais de linguagens de programação, escritas em semântica denotacional, de forma modular, para um interpretador em *Haskell* para essa linguagem. Pragmaticamente, a modularidade da descrição é guiada pela sintaxe abstrata da linguagem, onde cada módulo da especificação contém a descrição léxica, sintática e semântica de cada construção relevante da linguagem.

A compilação de definições escritas em módulos consiste em unir os componentes léxico, sintático e semântico da especificação para a geração de analisadores léxico, sintático e semântico para a linguagem especificada, que, agrupados, formam seu interpretador. Os problemas no processo de compilação surgem no momento da união dos componentes dos módulos para a geração de código, pois o projetista, ao escrever uma especificação modular, abstrai questões relacionadas à ordenação dos componentes, necessária para a geração de analisadores únicos da sintaxe e da semântica da linguagem especificada. Dessa maneira, esses problemas são resolvidos pelo compilador antes da geração de um interpretador para a especificação.

Este texto descreveu as técnicas e algoritmos implementados no compilador de *Notus* para geração do interpretador da linguagem a partir de especificações modulares. A compilação da especificação modular ocorre em duas etapas, a primeira, comum a qualquer compilador, reconhece a especificação léxica e sintaticamente; na segunda etapa, aplicam-se as técnicas e algoritmos apresentados para a geração do interpretador da linguagem. A segunda etapa do compilador é composta essencialmente por três subetapas de compilação executadas por tradutores responsáveis por gerar os analisadores léxico, sintático e semântico da linguagem.

O tradutor léxico do compilador opera inicialmente sobre as macros da definição ordenando-as de forma a garantir que, no código gerado, toda macro seja definida antes de seu uso. Em seguida, esse tradutor ordena as expressões regulares que definem

os *tokens* da linguagem, evitando-se que lexemas sejam capturados indevidamente. Adicionalmente, um analisador léxico otimizado é gerado, quando possível, evitando-se a geração de estados no autômato de reconhecimento de *tokens* para palavras-chave da linguagem especificada.

O tradutor sintático analisa a gramática concreta especificada, gerando uma gramática simplificada, aplicando os algoritmos para a eliminação de variáveis que não geram *strings* e de símbolos não-alcançáveis. Esse tradutor gera, além da especificação da gramática concreta, *datatypes* que representam a gramática abstrata, definida, opcionalmente, pelo projetista ou obtida pelo compilador, a partir da gramática concreta. Um papel importante do tradutor é tratar uniformemente as regras de produção definidas no momento da declaração da variável de gramática e as definidas, por meio de extensões, em outros módulos da especificação.

O tradutor semântico é responsável por gerar funções *Haskell*, a partir das definições de funções em *Notus*, mantendo a correspondência entre elas. *Haskell* processa as definições de uma função na ordem em que aparecem no código-fonte. Dessa maneira, o tradutor deve ordená-las para que não se sobreponham no código gerado. O sistema de tipos e subtipos de *Notus* é traduzido para *datatypes* em *Haskell*, via injeção e projeção de valores, de forma que um subtipo seja aceito onde seu supertipo é esperado e vice-versa. Como mostrado na Seção 6.4.1, a tradução de funções *Notus*, com domínios primitivos, não gera *overhead*, de forma que não há diferenças de tempo de execução significativas entre o código *Haskell* gerado pelo compilador e um código escrito diretamente em *Haskell*.

A utilização de programação orientada por aspectos na construção do compilador *Notus* permitiu a modularização dos algoritmos de análise semântica da especificação e da geração de código, por meio de inserção estática de métodos nas classes de nodos de *AST*. Com isso, o código do compilador *Notus* é modular e extensível, permitindo fácil modificação das etapas de análise semântica e geração de código já implementadas para novos nodos de *AST*, bem como inclusão de novas etapas.

O compilador *Notus* foi testado com especificações modulares em semântica denotacional para linguagens experimentais, escritas incrementalmente. As linguagens especificadas foram *Nano*, definida no manual da linguagem *Notus* [Tirelo e Bigonha, 2006], e *Tiny* e *Small*, definidas em [Gordon, 1979]. O conjunto de construções apresentadas por essas linguagens é comum à maioria das linguagens de programação reais. A complexidade inerente a especificações de linguagens de grande porte pode ser controlada por meio de sua decomposição em módulos, aliada a um ambiente de execução que permite a verificação experimental da correção da descrição. A execução da especificação formal das linguagens de teste permitiu a identificação de problemas nas equações semânticas de construtos da linguagem.

De forma simplificada, as principais contribuições deste trabalho são:

- criação de um compilador para a linguagem *Notus*, que produz interpretadores a partir de linguagens especificadas em semântica denotacional, permitindo prototipar linguagens de programação;
- aplicação de técnicas e algoritmos para a geração de analisadores léxico e sintático a partir de definições modulares;
- desenvolvimento de técnicas de compilação para manter a correspondência da tradução das definições de funções semânticas realizadas de forma modular em *Notus*, para a forma como *Haskell* as processa.

O compilador *Notus* apresentado neste trabalho suporta a escrita incremental das definições das partes léxica e sintática de uma linguagem de programação. No entanto, a escrita incremental da semântica depende da implementação da extensão do compilador para suporte à transformação de módulos e propagação de contexto, descrita nas Seções 3.8 e 3.9. A Seção 5.10 discute alternativas para a extensão do compilador para suporte à transformação de módulos. Etapas para a análise semântica e geração de código devem ser implementadas para as construções de propagação de contexto. Como trabalhos futuros, enumera-se:

- extensão do compilador desenvolvido para implementação das novas construções da linguagem *Notus*, descritas na Seção 3.8, para permitir extensibilidade e escalabilidade da definição da semântica de linguagens de programação;
- extensão do compilador para suporte à transformação de módulos como discutido na Seção 5.10;
- definição formal em *Notus* de linguagens de grande porte
- investigação para implementação mais eficiente da função pré-definida `update` do compilador *Notus*;
- construção de um ambiente interativo para desenvolvimento de especificações em *Notus*;
- especificação formal do sistema de tipos de *Notus* e do algoritmo de inferência de tipos da linguagem;
- análise comparativa do algoritmo de inferência de tipos implementado na atual versão do compilador *Notus* com obtido via formalização do sistema de tipos;
- definição formal do processo de geração de funções semânticas na forma de um sistema de reescrita.

Apêndice A

Executando o Compilador *Notus*

O compilador *Notus* foi desenvolvido na linguagem *Java*, e possui *Haskell* como linguagem alvo. Para execução do compilador *Notus* e dos interpretadores gerados são necessários os seguintes programas:

- ambiente de execução *Java JRE 5.0* disponível em `http://java.com/en/download/index.jsp`.
- gerador de analisadores léxicos para *Haskell Alex*, versão *alex-2.0.1* disponível em `http://www.haskell.org/alex`;
- gerador de analisadores sintáticos para *Haskell Happy*, versão *happy-1.15* disponível em `http://www.haskell.org/happy/`;
- compilador de *Haskell GHC*, versão *ghc-6.6* disponível em `http://www.haskell.org/ghc/`;

Existem versões para *Windows* e *Linux* de todos estes programas, portanto o compilador *Notus* pode ser executado em ambos ambientes.

O compilador *Notus* e as bibliotecas usadas por este estão disponíveis em `www.dcc.ufmg.br/~tays` em um arquivo compactado composto por uma pasta *lib* contendo as bibliotecas e o arquivo *jar* executável *notus.jar*. Os arquivos presentes na pasta *lib*: *aspctjrt.jar*; automaton.jar; cup_JLex.jar, assim como o arquivo *notus.jar*, devem ser colocados no *classpath*. Para executar uma especificação no compilador *Notus*, faça na linha de comando:

```
java notus.Notus <src> <out>
```

onde <src> é a pasta que contém o módulo *Main* da especificação e <out> é a pasta onde o interpretador será gerado. Em seguida, execute os geradores *Alex* e *Happy* para os arquivos gerados na pasta <out>, *Lexer.x* e *Syntactic.y*, e o *ghc* para o arquivo *Main.hs*:

```
alex Lexer.x
happy Syntactic.y
ghc --make -o <interpreter name> Main
```

O intepretador pode então ser executado na linha de comando e o primeiro argumento é um programa na linguagem especificada a ser interpretado, seguido, opcionalmente, dos argumentos de entrada e saída de acordo com a especificação da linguagem (veja Seção 3.2).

Apêndice B

A Linguagem Nano

B.1 Definição da Linguagem

Esta seção apresenta os módulos que compõem a especificação de *Nano*. Um programa em *Nano* é iniciado pela leitura de um valor para uma variável, seguida pela execução de um comando, e finalizado com a escrita do valor de uma expressão. Os comandos existentes em *Nano* são os de atribuição, condicional e repetição. As expressões em *Nano* são números, booleanos, aritméticas, relacionais e lógicas.

B.1.1 Kernel

O módulo `Kernel` na Listagem B.4 define que espaços em branco e comentários são ignorados pelo analisador léxico.

```
1 module Kernel
2
3 element layoutchar = " " | "\n" | "\t" | "\r" | "\f";
4 element comment = "//" .*;
5 ignore layoutchar | comment;
6
7 end
```

Listagem B.1: Módulo `Kernel` da definição de *Nano*

B.1.2 Expressões

O módulo de expressões de *Nano* da Listagem B.2 define todos os componentes léxico, sintático e semântico de expressões, como os domínios, *tokens*, gramáticas concreta e abstrata e as equações semânticas que descrevem a denotação dessas expressões.

```
1  module Expressions
2
3  token id = [a-zA-Z_][a-zA-Z0-9_]*;
4  token num : Value = [0-9]+ is asInteger;
5  token bool : Value = "True" | "False" is asBoolean;
6
7  Ans = Value | {error};
8
9  exp ::= exp "or" andexp | andexp : andexp;
10
11 andexp : Exp ::= andexp "and" relexp | relexp : relexp;
12
13 relexp : Exp ::= relexp "<" addexp | addexp : addexp;
14
15 addexp : Exp ::= addexp "+" multexp | multexp : multexp;
16
17 multexp : Exp ::= multexp "*" unexp     | unexp : unexp;
18
19 unexp : Exp ::= "not" primary | primary : primary;
20
21 primary : Exp ::= id | value :value;
22
23 value ::= bool : bool | num : num;
24
25 function dexp : Exp -> State -> Ans;
26 dexp bool state = bool;
27 dexp 0 state = 0;
28 dexp 1 state = 1;
29 dexp [id] state = state id;
30 dexp ["not" exp] state = let {e = dexp exp state } in
31         case e of { bool -> not bool;
32                     _   -> error};
33
34 dexp [exp1 "*" exp2] state =
35   let {
36   e1 = dexp exp1 state;
37   e2 = dexp exp2 state} in
38     case e1 of{
39       int1 -> case e2 of {
40                 int2 -> int1 * int2;
```

```
41                  _   -> error };
42        _   -> error};
43
44  dexp [exp1 "+" exp2] state = let {
45    e1 = dexp exp1 state;
46    e2 = dexp exp2 state} in
47      case e1 of{
48        int1 -> case e2 of {
49                  int2 -> int1 + int2;
50                   _   -> error };
51        _   -> error};
52
53
54  dexp [exp1 "<" exp2] state = let {
55    e1 = dexp exp1 state;
56    e2 = dexp exp2 state} in
57      case e1 of{
58        int1 -> case e2 of {
59                  int2 -> int1 < int2;
60                   _   -> error };
61        _   -> error};
62
63  dexp [exp1 "and" exp2] state = let {
64    e1 = dexp exp1 state;
65    e2 = dexp exp2 state} in
66        case e1 of{
67          bool1 -> case e2 of {
68                     bool2 -> bool1 and bool2;
69                      _   -> error };
70            _   -> error};
71
72  dexp [exp1 "or" exp2] state = let {
73    e1 = dexp exp1 state;
74    e2 = dexp exp2 state} in
75      case e1 of{
76        bool1 -> case e2 of {
77                   bool2 -> bool1 and bool2;
78                    _   -> error };
79          _   -> error};
80
```

```
81 end
```

Listagem B.2: Módulo `Expressions` da definição de *Nano*

O módulo `NewExpressions` da Listagem B.3 adiciona novas expressões à especificação de *Nano* a partir da extensão da gramática de expressões do módulo `Expressions`. Além disso, o módulo `NewExpressions` define as equações semânticas para essas expressões criando novas definições para a função de denotação de expressão `dexp`.

```
1  module NewExpressions
2
3  import Expressions;
4
5  extend relexp with relexp ">" addexp;
6
7  extend addexp with addexp "-" multexp;
8
9  extend multexp with multexp "/" unexp;
10
11 extend unexp with "-" primary;
12
13 extend primary with "(" exp ")":exp;
14
15 dexp int state = int;
16
17 dexp [exp1 ">" exp2] state = let {
18     e1 = dexp exp1 state;
19         e2 = dexp exp2 state} in
20       case e1 of{
21         int1 -> case e2 of {
22                              int2 -> int1 > int2;
23                                        _   -> error };
24           _   -> error};
25
26 dexp [exp1 "-" exp2] state = let {
27   e1 = dexp exp1 state;
28         e2 = dexp exp2 state} in
29     case e1 of{
30       int1 -> case e2 of {
31                           int2 -> int1 - int2;
32                  _   -> error };
33        _   -> error};
34
35 dexp [exp1 "/" exp2] state = let {
36   e1 = dexp exp1 state;
37   e2 = dexp exp2 state} in
38     case e1 of{
39       int1 -> case e2 of {
40                                int2 -> int1 / int2;
41                                          _   -> error };
42        _   -> error};
43
44 dexp ["-" exp] state = let {e = dexp exp state } in
45         case e of { int -> - int;
46                        _   -> error};
47 end
```

Listagem B.3: Módulo `NewExpressions` da definição de *Nano*

B.1.3 Comandos

O módulo de comandos de *Nano* da Listagem B.2 define todos os componentes léxico, sintático e semântico de comandos, como os domínios, *tokens*, gramáticas concreta e abstrata e as equações semânticas que descrevem a denotação desses comandos.

```
1  module Commands
2  import Expressions;
3
4  comm ::=
5    id ":=" exp
6  | "if" exp "then" comm1 "else" comm2:["ifthenelse" exp comm1 comm2]
7  | "while" exp "do" comm:["while" exp comm]
8  | "begin" comm+—";" "end";
9
10 Loc = Id;
11
12 State = Loc -> Value;
13
14 function fun : Loc -> Value;
15 fun id = 0;
16
17 function dcomm : Comm -> State -> State;
18 dcomm [id ":=" exp] state =
19   let {ans = dexp exp state} in
20     case ans of {
21       value -> state[id<-value];
22        _    -> state};
23
24 dcomm ["ifthenelse" exp comm1 comm2] state =
25   let {ans = dexp exp state} in
26     case ans of{
27       bool -> (if bool
28                  then (dcomm comm1)
29                  else (dcomm comm2)) state;
30        _   -> state};
31
32 dcomm ["while" exp comm] state =
33   let {ans = dexp exp state} in
34     case ans of{
35       bool ->
36         if bool
37           then (dcomm ["while" exp comm] . dcomm comm) state
38           else state;
39       _   -> state};
40
41 dcomm ["begin" comm+ "end"] state = dcommaux comm+ state;
42
43 function dcommaux : Comm* -> State -> State;
44 dcommaux (comm) = dcomm comm;
45 dcommaux c:comm* state =
46   let {s = dcomm c state} in
47     dcommaux comm* s;
```

Listagem B.4: Módulo `Commands` da definição de *Nano*

B.1.4 Módulo Principal

O módulo principal na Listagem B.5 define a sintaxe para um programa em *Nano*, e a equação semântica de denotação desse programa. A cláusula `syntax` define que a variável de gramática `prog` é o símbolo inicial da gramática concreta de *Nano*. A cláusula `semantics` define a função de avaliação da *AST* responsável por iniciar a interpretação do programa fonte. A cláusula `input` define que os dados de entrada serão lidos na entrada padrão, e a cláusula `output` define que os dados de saída serão escritos na saída padrão. Além disso os módulos `Expressions, NewExpressions, Commands` e `Kernel` são importados.

```
1  module Main
2  import Expressions, NewExpressions, Commands, Kernel;
3
4  syntax prog;
5  semantics dprog;
6
7  input stdin;
8  output stdout;
9
10 prog ::= "read" id ";" comm ";" "write" exp :["prog" id comm exp];
11
12 function dprog : Prog -> String -> String;
13 dprog ["prog" id comm exp] s =
14   evaluate { ansToString;
15              dexp exp;
16              dcomm comm fun[id<-(nextInt s)]};
17
18 function ansToString: Ans -> String;
19 ansToString ans =
20   case ans of {
21     int -> integerToString int;
22                 bool -> booleanToString bool;
23                  _   -> "program error"
24   };
```

Listagem B.5: Módulo `Main` da definição de *Nano*

B.2 Interpretador de *Nano*

O interpretador de *Nano* compreende os analisadores léxico e sintático, e o programa principal. As listagens contendo os arquivos gerados pelo compilador de *Notus* para a especificação de *Nano* são mostradas a seguir.

B.2.1 Analisador Léxico.

O analisador léxico de *Nano* é produzido pelo gerador de analisadores léxicos *Alex* a partir do arquivo de especificação léxica *Lexer.x* gerado pelo compilador de *Notus*. O arquivo *Lexer.x* é mostrado na Listagem B.6.

```
1  {
2  module Lexer where
3  import NotusDefault
4  import DataModule
5  import NotusFunctions
6  }
7
8  %wrapper "basic"
9
10 @layoutchar = " "|""[\n]""|""[\t]""|""[\r]""|""[\f]""
11 @comment = "//".*
12
13 tokens :-
14
15 "+"      { \s->(T__token9)}
16 ":="{ \s->(T__token18)}
17 "<"      { \s->(T__token8)}
18 "/"      { \s->(T__token14)}
19 "True"|"False"  { \s->(T__bool(Value32 (asBoolean s)))}
20 "*"      { \s->(T__token10)}
21 ")"      { \s->(T__token17)}
22 ";"      { \s->(T__token1)}
23 [a-zA-Z_][a-zA-Z0-9_]*  { \s->lookupT idTree s (T__id(Id29 s))}
24 "("              { \s->(T__token16)}
25 [0-9]+   { \s->(T__num(Value30 (asInteger s)))}
26 ">"      { \s->(T__token12)}
27 "-"      { \s->(T__token13)}
28
29 @layoutchar|@comment              ;
```

```
30
31 {
32 -- The token type:
33
34 data Token = T__token0
35             | T__token1
36             | T__token3
37             | T__token6
38             | T__token7
39             | T__token8
40             | T__token9
41             | T__token10
42             | T__token11
43             | T__token12
44             | T__token13
45             | T__token14
46             | T__token16
47             | T__token17
48             | T__token18
49             | T__token19
50             | T__token20
51             | T__token21
52             | T__token23
53             | T__token24
54             | T__token26
55             | T__token28
56             | T__id Id
57             | T__num Value
58             | T__bool Value
59          deriving Eq
60
61
62 idTree = (Node ("not",(T__token11))(Node ("else",(T__token21))
63           (Node ("begin",(T__token26))(Leaf ("and",(T__token7)))
64             (Leaf ("do",(T__token24))))
65           (Node ("end",(T__token28)) Null
66             (Leaf ("if",(T__token19)))) )
67           (Node ("then",(T__token20))
68           (Node ("read",(T__token0))
69             (Leaf ("or",(T__token6))) Null )
```

```
70          (Node ("while",(T__token23)) Null
71            (Leaf ("write",(T__token3)))) ))
72 }
```

Listagem B.6: Arquivo de especificação léxica *Lexer.x* para o *Alex*

B.2.2 Analisador Sintático

O analisador sintático é produzido pelo gerador de analisadores sintáticos *Happy* a partir do arquivo de especificação sintática *Syntactic.y* gerado pelo compilador de *Notus*. O arquivo *Syntactic.y* é mostrado na Listagem B.7.

```
1  {
2  module Syntactic where
3  import Char
4  import Lexer
5  import NotusDefault
6  import DataModule
7  import NotusFunctions
8  }
9
10 %name parse prog
11 %tokentype { Token }
12
13 %token
14         "read"                  { T__token0 }
15         ";"                     { T__token1 }
16         "write"                 { T__token3 }
17         id                      { T__id $$ }
18         num                     { T__num $$ }
19         bool                    { T__bool $$ }
20         "or"                    { T__token6 }
21         "and"                   { T__token7 }
22         "<"                     { T__token8 }
23         "+"                     { T__token9 }
24         "*"                     { T__token10 }
25         "not"                   { T__token11 }
26         ">"                     { T__token12 }
27         "-"                     { T__token13 }
28         "/"                     { T__token14 }
29         "("                     { T__token16 }
```

```
30          ")"                     { T__token17 }
31          ":="                    { T__token18 }
32          "if"                    { T__token19 }
33          "then"                  { T__token20 }
34          "else"                  { T__token21 }
35          "while"                 { T__token23 }
36          "do"                    { T__token24 }
37          "begin"                 { T__token26 }
38          "end"                   { T__token28 }
39 %%
40 prog     : "read" id ";" comm ";" "write" exp  { Prog34 $2 $4 $7 }
41 exp      : exp "or" andexp  { Exp35 $1 $3 }
42          | andexp  {  $1 }
43 andexp     : andexp "and" relexp  { Exp36 $1 $3 }
44            | relexp              {  $1 }
45 relexp     : relexp "<" addexp  { Exp37 $1 $3 }
46            | addexp              {  $1 }
47            | relexp ">" addexp   { Exp38 $1 $3 }
48 addexp     : addexp "+" multexp  { Exp39 $1 $3 }
49            | multexp             {  $1 }
50            | addexp "-" multexp   { Exp40 $1 $3 }
51 multexp     : multexp "*" unexp  { Exp41 $1 $3 }
52             | unexp              {  $1 }
53             | multexp "/" unexp  { Exp42 $1 $3 }
54 unexp     : "not" primary  { Exp43 $2 }
55           | primary  {  $1 }
56           | "-" primary  { Exp44 $2 }
57 primary     : id   { Exp45 $1 }
58             | value { ignoreIndirectionFun93 $1  }
59             | "(" exp ")"  {  $2 }
60 value     : bool  {  $1 }
61           | num  {  $1 }
62 comm     : id ":=" exp  { Comm46 $1 $3 }
63          | "if" exp "then" comm "else" comm  { Comm47 $2 $4 $6 }
64          | "while" exp "do" comm  { Comm48 $2 $4 }
65          | "begin" v92v "end"  { Comm49 $2 }
66 v92v              : v92v ";" comm  {($1++[$3])}
67 v92v              : comm  {[$1]}
68
69 {
```

```
70
71 happyError :: [Token] -> a
72 happyError _ = error "Parse error"
73
74
75 ----------------------Ignore Indirections Functions----------------------
76
77 ignoreIndirectionFun93 :: Value -> Exp
78 ignoreIndirectionFun93 igInd=
79         case igInd of
80                 (Value30 a1) -> (Exp50 a1)
81                 (Value32 a1) -> (Exp51 a1)
82
83 }
```

Listagem B.7: Arquivo de especificação sintática *Syntactic.y* para o *Happy*

B.2.3 Gramática Abstrata

A Listagem B.8 mostra o módulo `DataModule`, gerado pelo compilador, com os *datatypes* que representam a gramática abstrata de *Nano*.

```
1  module DataModule where
2
3  data Id = Id29 String
4          deriving Eq
5
6  data Value = Value30 Int
7             | Value32 Bool
8          deriving Eq
9
10 data Prog = Prog34 Id Comm Exp
11         deriving Eq
12
13 data Exp = Exp35 Exp Exp
14          | Exp36 Exp Exp
15          | Exp37 Exp Exp
16          | Exp38 Exp Exp
17          | Exp39 Exp Exp
18          | Exp40 Exp Exp
19          | Exp41 Exp Exp
20          | Exp42 Exp Exp
21          | Exp43 Exp
22          | Exp44 Exp
23          | Exp45 Id
24          | Exp50 Int
25          | Exp51 Bool
26         deriving Eq
27
28 data Comm = Comm46 Id Exp
29           | Comm47 Exp Comm Comm
30           | Comm48 Exp Comm
31           | Comm49 [Comm]
32         deriving Eq
```

Listagem B.8: Módulo `DataModule` contendo os *datatypes* da gramática abstrata de *Nano*

B.2.4 Programa Principal

O programa principal é composto pelas funções semânticas da especificação de *Nano*, e pela função principal main que inicia a execução do interpretador. A Listagem B.9 mostra o programa principal gerado para a especificação de *Nano*.

```
1  module Main where
2
3  import Lexer
4  import Syntactic
5  import DataModule
6  import NotusDefault
7  import NotusFunctions
8  import Data.Bits
9  import System
10
11 data Ans = U94 Value
12           | E95 Enum5
13         deriving Eq
14
15 data Loc = D96 Id
16         deriving Eq
17
18 data State = F97  (Loc -> Value)
19
20 data Enum5 = Enum5__Error
21         deriving Eq
22
23
24 ---------- Functions ----------
25
26 dprog :: (Prog -> (String -> String))
27 dprog =
28   ( \ ( (Prog34 id98 comm99 exp100 ) ) ->
29     ( (\string191 ->
30       ( ansToString ( ( dexp exp100 ) ( ( dcomm comm99 )
31         ( (F97 (update fun
32           [ ( (D96 id98  ) ) ] [ ( (Value30
33             ( ( nextInt string191 ) )  ) ) ]
34   )  ) ) ) ) ) ) )))
35
```

```
36 ansToString :: (Ans -> String)
37 ansToString = ( (\ans198-> case  ans198 of{
38  ((U94(Value30 int200))) -> ( integerToString int200 ) ;
39  ((U94(Value32 bool202))) -> ( booleanToString bool202 ) ;
40  _ -> "erro na avaliacao do programa" } ))
41
42 pp :: (String -> String)
43 pp = ( (\string203-> string203 ))
44
45 dexp :: (Exp -> (State -> Ans))
46 dexp = ( (\exp204-> case exp204 of {
47 (Exp51 bool110) ->( (\state205 ->((U94(Value32 (bool110 ))))));
48 (Exp50 0) ->( (\state207 ->((U94(Value30 (0 ))))));
49 (Exp50 1) ->( (\state209 ->((U94(Value30 (1 ))))));
50 (Exp50 int159) ->( (\state211 ->((U94(Value30 (int159 ))))));
51 (Exp45 id114 ) ->( (\state213 ->
52   ((U94 (( (case state213 of ((F97 funExpressions215)) ->
53     funExpressions215) ( (D96 id114  ) ) ) )))));
54 (Exp43 exp120 ) ->( (\state216 ->let {
55 a0 = ( ( dexp exp120 ) state216 ) }in  case  a0 of{
56  ((U94(Value32 bool219))) ->
57    ( (U94 (Value32 ( not  bool219 )  ) ) ) ;
58  _ -> ( (E95 (Enum5__Error  ) ) ) } ));
59 (Exp41 exp124 exp125 ) ->( (\state220 ->let {
60 a0 = ( ( dexp exp124 ) state220 ) ;
61 a1 = ( ( dexp exp125 ) state220 ) }in  case  a0 of{
62  ((U94(Value30 int224))) -> case  a1 of{
63  ((U94(Value30 int225))) ->
64    ( (U94 (Value30 ( ( int224 ) * ( int225 ) )  ) ) ) ;
65  _ -> ( (E95 (Enum5__Error  ) ) ) } ;
66  _ -> ( (E95 (Enum5__Error  ) ) ) } ));
67 (Exp39 exp147 exp148 ) ->( (\state226 ->let {
68 a0 = ( ( dexp exp147 ) state226 ) ;
69 a1 = ( ( dexp exp148 ) state226 ) }in  case  a0 of{
70  ((U94(Value30 int230))) -> case  a1 of{
71  ((U94(Value30 int231))) ->
72    ( (U94 (Value30 ( ( int230 ) + ( int231 ) )  ) ) ) ;
73  _ -> ( (E95 (Enum5__Error  ) ) ) } ;
74  _ -> ( (E95 (Enum5__Error  ) ) ) } ));
75 (Exp37 exp150 exp151 ) ->( (\state232 ->let {
```

```
76  a0 = ( ( dexp exp150 ) state232 ) ;
77  a1 = ( ( dexp exp151 ) state232 ) }in  case  a0 of{
78   ((U94(Value30 int236))) -> case  a1 of{
79   ((U94(Value30 int237))) ->
80     ( (U94 (Value32 ( ( int236 ) < ( int237 ) )  ) ) ) ;
81   _ -> ( (E95 (Enum5__Error  ) ) ) } ;
82   _ -> ( (E95 (Enum5__Error  ) ) ) } ));
83  (Exp36 exp153 exp154 ) ->( (\state238 ->let {
84  a0 = ( ( dexp exp153 ) state238 ) ;
85  a1 = ( ( dexp exp154 ) state238 ) }in  case  a0 of{
86   ((U94(Value32 bool242))) -> case  a1 of{
87   ((U94(Value32 bool243))) ->
88   ( (U94 (Value32 ( ( bool242 ) && ( bool243 ) )  ) ) ) ;
89   _ -> ( (E95 (Enum5__Error  ) ) ) } ;
90   _ -> ( (E95 (Enum5__Error  ) ) ) } ));
91  (Exp35 exp156 exp157 ) ->( (\state244 ->let {
92  a0 = ( ( dexp exp156 ) state244 ) ;
93  a1 = ( ( dexp exp157 ) state244 ) }in  case  a0 of{
94   ((U94(Value32 bool248))) -> case  a1 of{
95   ((U94(Value32 bool249))) ->
96     ( (U94 (Value32 ( ( bool248 ) && ( bool249 ) )  ) ) ) ;
97   _ -> ( (E95 (Enum5__Error  ) ) ) } ;
98   _ -> ( (E95 (Enum5__Error  ) ) ) } ));
99  (Exp38 exp161 exp162 ) ->( (\state250 ->let {
100 a0 = ( ( dexp exp161 ) state250 ) ;
101 a1 = ( ( dexp exp162 ) state250 ) }in  case  a0 of{
102  ((U94(Value30 int254))) -> case  a1 of{
103  ((U94(Value30 int255))) ->
104    ( (U94 (Value32 ( ( int254 ) > ( int255 ) )  ) ) ) ;
105  _ -> ( (E95 (Enum5__Error  ) ) ) } ;
106  _ -> ( (E95 (Enum5__Error  ) ) ) } ));
107 (Exp40 exp164 exp165 ) ->( (\state256 ->let {
108 a0 = ( ( dexp exp164 ) state256 ) ;
109 a1 = ( ( dexp exp165 ) state256 ) }in  case  a0 of{
110  ((U94(Value30 int260))) -> case  a1 of{
111  ((U94(Value30 int261))) ->
112    ( (U94 (Value30 ( ( int260 ) - ( int261 ) )  ) ) ) ;
113  _ -> ( (E95 (Enum5__Error  ) ) ) } ;
114  _ -> ( (E95 (Enum5__Error  ) ) ) } ));
115 (Exp42 exp167 exp168 ) ->( (\state262 ->let {
```

```
116 a0 = ( ( dexp exp167 ) state262 ) ;
117 a1 = ( ( dexp exp168 ) state262 ) }in  case  a0 of{
118  ((U94(Value30 int266))) -> case  a1 of{
119  ((U94(Value30 int267))) ->
120    ( (U94 (Value30 ( div  ( int266 ) ( int267 ) )  ) ) ) ;
121  _ -> ( (E95 (Enum5__Error  ) ) ) } ;
122  _ -> ( (E95 (Enum5__Error  ) ) ) } ));
123 (Exp44 exp170 ) ->( (\state268 ->let {
124 a0 = ( ( dexp exp170 ) state268 ) }in  case  a0 of{
125  ((U94(Value30 int271))) -> ( (U94 (Value30
126    ( negate  int271 )  ) ) ) ;
127  _ -> ( (E95 (Enum5__Error  ) ) ) } ))
128 }))
129
130 fun :: (Loc -> Value)
131 fun = ( \( ((D96 id172)) ) ->((Value30 (0 ))))
132
133 dcomm :: (Comm -> (State -> State))
134 dcomm = ( (\comm273-> case comm273 of {
135 (Comm46 id173 exp174 ) ->( (\state274 ->let {
136 ans276 = ( ( dexp exp174 ) state274 ) }in  case  ans276 of{
137  ((U94 a3)) ->
138    ( (F97 (update (case state274 of
139      ((F97 funExpressions278)) -> funExpressions278)
140        [ ( (D96 id173  ) ) ] [ a3 ])  ) ) ;
141       _ -> state274 } ));
142 (Comm47 exp176 comm177 comm178 ) ->( (\state279 ->let {
143 ans281 = ( ( dexp exp176 ) state279 ) }in  case  ans281 of{
144  ((U94(Value32 bool283))) ->
145    ( ( if ( bool283 )
146      then ( ( ( dcomm comm177 ) ) )
147        else ( ( ( dcomm comm178 ) ) ) ) state279 ) ;
148    _ -> state279 } ));
149 (Comm48 exp180 comm181 ) ->( (\state286 ->let {
150 ans288 = ( ( dexp exp180 ) state286 ) }in  case  ans288 of{
151  ((U94(Value32 bool290))) ->
152    if ( bool290 )
153      then ( ( ( ( \x293-> ( dcomm ( (Comm48 exp180 comm181  ) ) )
154        ( ( dcomm comm181 ) x293  ) ) ) state286 ) )
155      else ( state286 ) ;
```

```
156   _ -> state286 } ));
157 (Comm49 comm183 ) ->((\state294 ->( ( dcommaux comm183 ) state294)))
158 }))
159
160 dcommaux :: ([Comm] -> (State -> State))
161 dcommaux = ( (\comm297-> case comm297 of {
162 ( [comm186]) ->( (\state298 ->( dcomm comm186 ) state298 ));
163 ( (comm301:comm187)) ->( (\state302 ->let {
164 a0 = ( ( dcomm comm301 ) state302 ) }in
165   ( ( dcommaux comm187 ) a0 ) ))
166 }))
167
168 main = do
169   (x:xs) <- getArgs
170   progL <- readFile x
171   inL <- sReaderNotus (head(matchInFiles [(Stdin)] xs))
172   sWriterNotus (head(matchOutFiles [(Stdout)] xs))
173     (dprog  (parse (alexScanTokens progL)) inL)
```

Listagem B.9: Módulo principal do interpretador *Nano*

Apêndice C

A Linguagem Tiny

C.1 Definição da Linguagem

Esta seção apresenta os módulos que compõem a especificação de *Tiny*. Um programa em *Tiny* é formado por um ou mais comandos que podem conter expressões. Os comandos existentes em *Tiny* são os de atribuição, condicional, repetição, saída e seqüência de comandos. As expressões em *Tiny* são números, booleanos, aritmética de adição, igualdade, negação lógica e leitura.

C.1.1 Kernel

O módulo `Kernel`, o mesmo usado na definição de *Nano* mostrado na Listagem B.4, define que espaços em branco e comentários são ignorados pelo analisador léxico.

C.1.2 Expressões

As expressões de *Tiny* foram definidas no pacote `Expressions` que contém o módulo `CoreExps`, que define as expressões originais da Linguagem *Tiny*, e o módulo `NewExpressions` que estende o módulo `Expressions` definindo novas expressões.

O módulo de expressões de *Tiny* da Listagem C.1 define todos os componentes léxico, sintático e semântico de expressões, como os domínios, *tokens*, gramáticas concreta e abstrata e as equações semânticas que descrevem a denotação dessas expressões.

```
1 module Expressions.CoreExps
2
3 import Domains, Auxiliary, Semantics;
4
5 public syntactic Exp, Value, Ide;
6
```

```
7  token id    : Ide   = [a-zA-Z_][a-zA-Z0-9_]*;
8  token num   : Value  = [0-9]+ is asInteger;
9  token bool  : Value = "True" | "False" is asBoolean;
10
11 e : Exp ::= e "=" binop | binop : binop ;
12 binop : Exp ::= binop "+" unop | unop : unop;
13 unop : Exp ::= "not" "(" e ")"| primary : primary;
14 primary : Exp ::= id | "read" | value:value;
15 value ::= num:num | bool:bool;
16
17 // Semantic clauses
18 // (expressions)
19 de ["read"] econt State(m,i:is*,o) = econt i State(m,is*,o);
20 //de ["read"] econt State(m,(),o) = econt 0 State(m,(),o);
21 de int econt = econt int;
22 de bool econt = econt bool;
23 de [id] econt State(m,i,o) =
24   case m id of
25     {unbound -> error;
26      value -> econt value State(m,i,o)};
27 de [e1 "=" e2] econt =
28   (de e1 (\v1 ->
29     de e2 (\v2 -> econt (equals v1 v2))));
30 de ["not" "(" e ")"] econt =
31   de e (\value state -> (case value of {
32     bool -> econt (not bool) state;
33      _   -> error}));
34 de [e1 "+" e2] econt =
35   de e1 (\v1 ->
36     de e2 (\v2 ->
37       case v1 of {int1 -> case v2 of {int2 -> econt (int1 + int2);
38                                       _   -> err};
39                   _   -> err}));
40 end
```

Listagem C.1: Módulo `CoreExps` da definição de *Tiny*

O módulo `NewExpressions` da Listagem C.2 adiciona novas expressões à especificação de *Notus* a partir da extensão da gramática de expressões do módulo `Expressions`. Além disso, o módulo `NewExpressions` define as equações semânticas para estas expressões criando novas definições para a função de denotação de expressão `de`.

```
1  module Expressions.NewExpressions
2
3  import Domains, Semantics, Expressions.CoreExps;
4
5  extend e with e ">" binop;
6  extend e with e "<" binop;
7  extend e with e "-" binop;
8  extend e with e "mod" binop;
9  extend primary with "(" e ")":e;
10
11 de [e1 ">" e2] econt =
12   (de e1 (\v1 -> de e2
13     (\v2 -> if (v1 is int) and (v2 is int)
14                           then econt (gt v1 v2)
15               else err)));
16
17 de [e1 "<" e2] econt =
18   (de e1 (\v1 ->
19     de e2
20       (\v2 -> if (v1 is int) and (v2 is int)
21                 then econt (lt v1 v2)
22                 else err)));
23
24 de [e1 "-" e2] econt =
25   de e1 (\v1 ->
26     de e2 (\v2 ->
27       case v1 of {
28         int1 -> case v2 of {
29           int2 -> econt (int1 - int2);
30           _    -> err};
31                                 _    -> err}));
32
33 de [e1 "mod" e2] econt =
34   de e1 (\v1 ->
35     de e2 (\v2 ->
36       case v1 of {
37         int1 -> case v2 of {
38           int2 -> econt (int1 mod int2);
39           _    -> err};
40         _    -> err}));
41
42 function gt : Value -> Value -> Bool;
43 gt int1 int2 = (int1 > int2);
44 gt int _ = false;
45 gt _ int = false;
46
47 function lt : Value -> Value -> Bool;
48 lt int1 int2 = (int1 < int2);
49 lt int _ = false;
50 lt _ int = false;
51 end
```

Listagem C.2: Módulo `NewExpressions` da definição de *Tiny*

C.1.3 Comandos

O módulo de comandos de *Tiny* da Listagem C.3 define todos os componentes léxico, sintático e semântico de comandos, como os domínios, *tokens*, gramáticas concreta e abstrata e as equações semânticas que descrevem a denotação desses comandos.

```
1  module Commands
2  import Expressions;
3
4  module Commands.CoreComm
5
6  import Expressions.CoreExps, Domains, Auxiliary, Semantics;
7
8  public syntactic Com;
9
10 public c : Com ::= c ";" simpleCom
11                  | simpleCom : simpleCom;
12 public simpleCom : Com ::= id ":=" e
13                           | "output" e
14                           | "if" e "then" c1 "else" c2 "end"
15                           | "while" e "do" c "end" ;
16
17
18 // funcionam com cont mas nao com c
19 dc [id ":=" e] cont =
20   de e (\v State(m,i,o) -> cont State(m[id <- v],i,o));
21
22 dc ["output" e] cont =
23   de e (\v State(m,i,o) -> cont State(m,i,v:o));
24
25 dc ["if" e "then" c1 "else" c2 "end"] cont =
26   de e (\v -> case v of
27     {bool -> if bool
28                then dc c1 cont
29                else dc c2 cont;
30      _    -> err});
31 dc ["while" e "do" c "end"] cont =
32   de e (\v -> case v of
33     {bool -> if bool
34                then evaluate {dc c ;dc ["while" e "do" c "end"] cont}
35                else cont;
36      _    -> err});
37 dc [c1 ";" c2] cont = dc c1 (dc c2 cont);
```

Listagem C.3: Módulo `Commands` da definição de *Tiny*

O módulo `Commands.NewCommands` da Listagem C.4 adiciona o novo comando *do while* à especificação de *Notus* a partir da extensão da gramática de comandos do módulo `Commands.CoreComm`. Além disso, o módulo `NewCommands` define as equação semântica para este comando criando uma nova definição para a função de denotação de comandos `dc`.

```
1  module Commands.NewCommands
2
3  import Commands.CoreComm;
4
5  extend simpleCom with "do" c "while" e;
6
7  dc ["do" c "while" e] cont =
8    dc c (de e (\v -> case v of
9            {bool -> if bool
10                              then dc ["do" c "while" e] cont
11                   else cont;
12       _     -> err}));
```

Listagem C.4: Módulo `NewCommands` da definição de *Tiny*

C.1.4 Módulo Principal

O módulo principal na Listagem C.5 define a equação semântica de denotação de um programa em *Tiny*, além de funções auxiliares para tratamento da entrada e saída dos programas em *Tiny*. A cláusula `syntax` define que a variável de gramática `c` é o símbolo inicial da gramática concreta de *Tiny*. A cláusula `semantics` define a função de avaliação da *AST* responsável por iniciar a interpretação do programa fonte. A cláusula `input` define que os dados de entrada serão lidos na entrada padrão, e a cláusula `output` define que os dados de saída serão escritos na saída padrão.

```
1  module Main
2
3  import Domains, Semantics, Kernel, Expressions.CoreExps,
4         Expressions.NewExpressions, Commands.CoreComm,
5         Commands.NewCommands;
6
7  syntax c;
8  semantics dprog;
9
10 input stdin;
11 output stdout;
12
13 function dprog : Com -> String -> String;
14 dprog com i =
15   let {State(m,i1,o) =
16     dc com stateId State(\x -> unbound, evalInput i, ())}
17         in showOutput o;
18
19 function showOutput : Output -> String;
20 showOutput () = "";
21 showOutput v:vs* =
22   showOutput vs* ++
23   "\t" ++
24   (case v of {
25     int  -> integerToString int;
26                 bool -> booleanToString bool });
27 function evalInput : String -> Value*;
28 evalInput s =
29   (intToValue(nextInt s)):evalInput (ignoreInt s);
30
31 function intToValue : Int -> Value;
32 intToValue i = i;
33
34 function stateId : Cont;
35 stateId state = state;
```

Listagem C.5: Módulo `Main` da definição de *Tiny*

C.2 Interpretador de *Tiny*

O interpretador de *Tiny* compreende os analisadores léxico e sintático, e o programa principal. As listagens contendo os arquivos gerados pelo compilador de *Notus* para a especificação de *Tiny* são mostradas a seguir.

C.2.1 Analisador Léxico.

O analisador léxico de *Tiny* é produzido pelo gerador de analisadores léxicos *Alex* a partir do arquivo de especificação léxica *Lexer.x* gerado pelo compilador de *Notus*. O arquivo *Lexer.x* é mostrado na Listagem C.6.

```
1  {
2  module Lexer where
3  import NotusDefault
4  import DataModule
5  import NotusFunctions
6  }
7
8  %wrapper "basic"
9
10 @kernel__layoutchar = " "|""[\n]""|""[\t]""|""[\r]""|""[\f]""
11 @kernel__comment = "//".*
12
13 tokens :-
14
15 "True"|"False"
16   { \s->(T__coreexps__bool(Value__29 (asBoolean s)))}
17 [a-zA-Z_][a-zA-Z0-9_]*
18   { \s->lookupT coreexps__idTree s (T__coreexps__id(Ide__26 ( s)))}
19 ":="              { \s->(T__corecomm__token__15)}
20 "("               { \s->(T__coreexps__token__5)}
21 "+"               { \s->(T__coreexps__token__3)}
22 [0-9]+
23   { \s->(T__coreexps__num(Value__27 (asInteger s)))}
24 "<"               { \s->(T__newexpressions__token__9)}
25 ")"               { \s->(T__coreexps__token__6)}
26 ";"               { \s->(T__corecomm__token__14)}
27 "="               { \s->(T__coreexps__token__2)}
28 "-"               { \s->(T__newexpressions__token__10)}
29 ">"               { \s->(T__newexpressions__token__8)}
```

```
30
31 @kernel__layoutchar|@kernel__comment             ;
32
33 {
34 -- The token type:
35
36 data Token__Main = T__coreexps__token__2
37                  | T__coreexps__token__3
38                  | T__coreexps__token__4
39                  | T__coreexps__token__5
40                  | T__coreexps__token__6
41                  | T__coreexps__token__7
42                  | T__newexpressions__token__8
43                  | T__newexpressions__token__9
44                  | T__newexpressions__token__10
45                  | T__newexpressions__token__11
46                  | T__corecomm__token__14
47                  | T__corecomm__token__15
48                  | T__corecomm__token__16
49                  | T__corecomm__token__17
50                  | T__corecomm__token__18
51                  | T__corecomm__token__19
52                  | T__corecomm__token__20
53                  | T__corecomm__token__21
54                  | T__corecomm__token__22
55                  | T__coreexps__id Ide__CoreExps
56                  | T__coreexps__num Value__CoreExps
57                  | T__coreexps__bool Value__CoreExps
58          deriving Eq
59
60
61
62
63 data Tree a b = Leaf (a,b)
64          | Node (a,b) (Tree a b) (Tree a b)
65          | Null
66      deriving Eq
67
68 lookupT :: Ord a => Tree a Token__Main -> a -> Token__Main -> Token__Main
69 lookupT (Leaf (a,b)) s tReturn
```

```
70    | s == a = b
71    | otherwise = tReturn
72  lookupT (Node (n,t) l r) s tReturn
73    | n == s  = t
74    | s > n   = lookupT r s tReturn
75    | s < n   = lookupT l s tReturn
76  lookupT Null _ tReturn = tReturn
77
78  coreexps__idTree =
79    (Node ("not",(T__coreexps__token__4))
80    (Node ("end",(T__corecomm__token__20))
81    (Node ("else",(T__corecomm__token__19))
82      (Leaf ("do",(T__corecomm__token__22))) Null )
83    (Node ("if",(T__corecomm__token__17)) Null
84      (Leaf ("mod",(T__newexpressions__token__11)))) )
85    (Node ("then",(T__corecomm__token__18))
86    (Node ("read",(T__coreexps__token__7))
87      (Leaf ("output",(T__corecomm__token__16))) Null )
88      (Leaf ("while",(T__corecomm__token__21)))))
89  }
```

Listagem C.6: Arquivo de especificação léxica *Lexer.x* para o *Alex*

C.2.2 Analisador Sintático

O analisador sintático é produzido pelo gerador de analisadores sintáticos *Happy* a partir do arquivo de especificação sintática *Syntactic.y* gerado pelo compilador de *Notus*. O arquivo *Syntactic.y* é mostrado na Listagem C.7.

```
1  {
2  module Syntactic where
3  import Char
4  import Lexer
5  import NotusDefault
6  import DataModule
7  import NotusFunctions
8  }
9
10 %name parse corecomm__c
11 %tokentype { Token__Main }
12
```

```
13 %token
14         coreexps__id                    { T__coreexps__id $$ }
15         coreexps__num                   { T__coreexps__num $$ }
16         coreexps__bool                  { T__coreexps__bool $$ }
17         "="                     { T__coreexps__token__2 }
18         "+"                     { T__coreexps__token__3 }
19         "not"           { T__coreexps__token__4 }
20         "("                     { T__coreexps__token__5 }
21         ")"                     { T__coreexps__token__6 }
22         "read"  { T__coreexps__token__7 }
23         ">"                     { T__newexpressions__token__8 }
24         "<"                     { T__newexpressions__token__9 }
25         "-"                     { T__newexpressions__token__10 }
26         "mod"           { T__newexpressions__token__11 }
27         ";"                     { T__corecomm__token__14 }
28         ":="            { T__corecomm__token__15 }
29         "output"{ T__corecomm__token__16 }
30         "if"            { T__corecomm__token__17 }
31         "then"  { T__corecomm__token__18 }
32         "else"  { T__corecomm__token__19 }
33         "end"           { T__corecomm__token__20 }
34         "while" { T__corecomm__token__21 }
35         "do"            { T__corecomm__token__22 }
36 %%
37 coreexps__e : coreexps__e "=" coreexps__binop { Exp__31 $1 $3 }
38             | coreexps__binop {  $1 }
39             | coreexps__e ">" coreexps__binop { Exp__32 $1 $3 }
40             | coreexps__e "<" coreexps__binop { Exp__33 $1 $3 }
41             | coreexps__e "-" coreexps__binop { Exp__34 $1 $3 }
42             | coreexps__e "mod" coreexps__binop { Exp__35 $1 $3 }
43 coreexps__binop : coreexps__binop "+" coreexps__unop { Exp__36 $1 $3 }
44                 | coreexps__unop {  $1 }
45 coreexps__unop  : "not" "(" coreexps__e ")"     { Exp__37 $3 }
46                 | coreexps__primary {  $1 }
47 coreexps__primary : coreexps__id { Exp__38 $1 }
48                   | "read" { Exp__39 }
49                   | coreexps__value { ignoreIndirectionFun__70 $1  }
50                   | "(" coreexps__e ")" {  $2 }
51 coreexps__value : coreexps__num {  $1 }
52                 | coreexps__bool        {  $1 }
```

```
53 corecomm__c : corecomm__c ";" corecomm__simpleCom { Com__40 $1 $3 }
54             | corecomm__simpleCom {  $1 }
55 corecomm__simpleCom : coreexps__id ":=" coreexps__e { Com__41 $1 $3 }
56                    | "output" coreexps__e { Com__42 $2 }
57                    | "if" coreexps__e "then"
58                      corecomm__c "else"
59                      corecomm__c "end" { Com__43 $2 $4 $6 }
60                    | "while" coreexps__e
61                      "do" corecomm__c "end" { Com__44 $2 $4 }
62                    | "do" corecomm__c
63                      "while" coreexps__e { Com__45 $2 $4 }
64
65 {
66
67 happyError :: [Token__Main] -> a
68 happyError _ = error "Parse error"
69
70
71 ---------------------Ignore Indirections Functions---------------------
72
73 ignoreIndirectionFun__70 :: Value__CoreExps -> Exp__CoreExps
74 ignoreIndirectionFun__70 igInd=
75         case igInd of
76                 (Value__27 a1) -> (Exp__46 a1)
77                 (Value__29 a1) -> (Exp__47 a1)
78
79 }
```

Listagem C.7: Arquivo de especificação sintática *Syntactic.y* para o *Happy*

C.2.3 Gramática Abstrata

A Listagem C.8 mostra o módulo `DataModule`, gerado pelo compilador, com os *datatypes* que representam a gramática abstrata de *Tiny*.

```
1  module DataModule where
2
3
4  data Ide__CoreExps = Ide__26 String
5          deriving Eq
6
7  data Value__CoreExps = Value__27 Int
8                       | Value__29 Bool
9          deriving Eq
10
11 data Exp__CoreExps = Exp__31 Exp__CoreExps  Exp__CoreExps
12                    | Exp__32 Exp__CoreExps  Exp__CoreExps
13                    | Exp__33 Exp__CoreExps  Exp__CoreExps
14                    | Exp__34 Exp__CoreExps  Exp__CoreExps
15                    | Exp__35 Exp__CoreExps  Exp__CoreExps
16                    | Exp__36 Exp__CoreExps  Exp__CoreExps
17                    | Exp__37 Exp__CoreExps
18                    | Exp__38 Ide__CoreExps
19                    | Exp__39
20                    | Exp__46 Int
21                    | Exp__47 Bool
22         deriving Eq
23
24 data Com__CoreComm = Com__40 Com__CoreComm  Com__CoreComm
25                    | Com__41 Ide__CoreExps  Exp__CoreExps
26                    | Com__42 Exp__CoreExps
27                    | Com__43 Exp__CoreExps  Com__CoreComm
   Com__CoreComm
28                    | Com__44 Exp__CoreExps  Com__CoreComm
29                    | Com__45 Com__CoreComm  Exp__CoreExps
30         deriving Eq
```

Listagem C.8: Módulo `DataModule` contendo os *datatypes* da gramática abstrata de *Tiny*

C.2.4 Programa Principal

O programa principal é composto pelas funções semânticas da especificação de *Tiny*, e pela função principal main que inicia a execução do interpretador. A Listagem C.9 mostra o programa principal gerado para a especificação de *Tiny*.

```
1  module Main where
2
3  import Lexer
4  import Syntactic
5  import DataModule
6  import NotusDefault
7  import NotusFunctions
8  import Data.Bits
9  import System
10
11 data State = T71__State (Memory , Input , Output)
12
13
14 data MemoryValue = U72 Value
15                  | U73 Unbound
16         deriving Eq
17
18
19 data Memory = F74  (Ide -> MemoryValue)
20
21
22 data Input = L75 [ Value ]
23         deriving Eq
24
25
26 data Output = L76 [ Value ]
27         deriving Eq
28
29
30 data Ans = U77 State
31          | U78 Error
32
33
34 data Cont = F79  (State -> Ans)
35
```

```
36
37 data Econt = F80  (Value -> Cont)
38
39
40 data Error = E81 Enum0
41         deriving Eq
42
43
44 data Unbound = E82 Enum1
45         deriving Eq
46
47
48 data ValueStatePair = T83__ValueStateTuple (Value , State)
49                    | U84 Error
50
51
52 data Enum0 = Enum0__Error
53         deriving Eq
54
55
56 data Enum1 = Enum1__Unbound
57         deriving Eq
58
59
60 ---------- Functions ----------
61
62 dprog :: (Com -> (String -> String))
63 dprog =
64   ( (\com193-> ( (\string194 ->let {
65     (x196,x197,x198) = (case
66       ( (case ( ( dc com193 ) stateId ) of
67         ((F79 funDomains203)) -> funDomains203)
68           ((T71__State ( ( (F74 (\ a1 ->
69             ( (U73 (E82 (Enum1__Unbound  ) ) ) ) )  ) ) ,
70             ( (L75 ( evalInput string194 )  ) ) ,
71             ( (L76 [ ]  ) ) )  ) ) ) of
72               ((U77(T71__State state204))) -> state204) }
73      in  ( showOutput x198 ) ))))
74
75 showOutput :: (Output -> String)
```

```
76 showOutput =
77   ( (\output205-> case output205 of {
78   ((L76 [])) ->"" ;
79   ((L76 (value206:value207))) ->
80     ( ( ( showOutput ( (L76 value207  ) ) )  ++
81       "\t" )  ++
82       ( case  value206 of{
83         ((Value27 int210)) -> ( integerToString int210 ) ;
84         ((Value29 bool212)) -> ( booleanToString bool212 ) } ) )
85 }))
86
87 evalInput :: (String -> [Value])
88 evalInput =
89   ( (\string213->
90     ( ( ( intToValue ( ( nextInt string213 ) ) ) ) :
91       ( evalInput ( ( ignoreInt string213 ) ) ) ) ))
92
93 intToValue :: (Int -> Value)
94 intToValue = ( (\int218-> ((Value27 (int218 )))))
95
96 stateId :: Cont
97 stateId = (F79(\state219-> ((U77 (state219 )))))
98
99 de :: (Exp -> (Econt -> Cont))
100 de =
101   ( (\exp220-> case exp220 of {
102     (Exp39 ) ->
103       ( (\econt221 ->
104         (F79(
105           \((T71__State (( memory95),
106              ((L75 (value225:value226))),
107              ( output96)))) ->
108              ( (case
109                ( (case econt221 of
110                  ((F80 funDomains227)) -> funDomains227) value225 )
111                  of ((F79 funDomains228)) -> funDomains228)
112              ( (T71__State ( ( memory95  ) ,
113              ( (L75 value226  ) ) ,
114              ( output96  ) )  ) ) ) ))));
115
```

```
116     (Exp46 int97) ->
117       ( (\econt229 ->
118         (F79(\state231 ->
119           (case
120             ( (case econt229 of ((F80 funDomains234)) -> funDomains234)
121               ( (Value27 int97  ) ) ) of
122                 ((F79 funDomains235)) -> funDomains235) state231 ))));
123
124     (Exp47 bool99) ->
125       ( (\econt236 ->
126          (F79(\state238 ->
127           (case
128             ( (case econt236 of
129                 ((F80 funDomains241)) -> funDomains241)
130             ( (Value29 bool99  ) ) ) of
131               ((F79 funDomains242)) -> funDomains242) state238 ))));
132
133     (Exp38 ide101 ) ->
134       ( (\econt243 ->
135         (F79(\((T71__State (( memory103),( input104),( output105)))) ->
136           case  ( (case memory103 of
137             ((F74 funDomains249)) -> funDomains249) ide101 ) of{
138               ((U73 unbound247)) -> ( (U78 (E81 (Enum0__Error  ) ) ) ) ;
139             ((U72 value248)) ->
140               ( (case ( (case econt243 of ((F80 funDomains250)) ->
141                 funDomains250) value248 ) of
142                   ((F79 funDomains251)) -> funDomains251)
143                   ( (T71__State ( ( memory103  ) ,
144                   ( input104  ) ,
145                   ( output105  ) )  ) ) ) } ))));
146
147     (Exp31 exp106 exp107 ) ->
148       ( (\econt252 ->
149         (F79(\state254 ->
150           (case ( ( ( de exp106 )
151             ( (F80 ( (\ a0 -> ( ( de exp107 )
152             ( (F80 ( (\ a1 ->
153               ( (case econt252 of ((F80 funDomains260)) ->
154               funDomains260)
155               ( (Value29 ( ( ( equals a0 ) a1 ) )  ) ) ) ) ) )
```

```
156             ) ) ) ) )   ) ) ) ) ) of
157               ((F79 funDomains261)) -> funDomains261) state254 ))));
158
159     (Exp37 exp125 ) ->
160       ( (\econt262 ->
161         (F79(\state264 ->(case
162           ( ( de exp125 )
163             ( (F80 ( (\ value266 -> ( (F79 (\ state267 ->
164             ( case  value266 of{
165               ((Value29 bool269)) ->
166                 ( (case ( (case econt262 of
167                   ((F80 funDomains272)) -> funDomains272)
168                     ( (Value29 ( ( not  bool269 ) )  ) ) )
169                  of ((F79 funDomains273)) ->
170                   funDomains273) state267 ) ;
171                 _ -> ( (U78 (E81 (Enum0__Error  ) ) ) ) }
172               ) )  ) ) ) )  ) ) )
173                 of ((F79 funDomains274)) ->
174                   funDomains274) state264 ))));
175
176      (Exp36 exp130 exp131 ) ->
177        ( (\econt275 ->(F79(\state277 ->
178          (case ( ( de exp130 )
179            ( (F80 ( (\ a0 -> ( ( de exp131 )
180              ( (F80 ( (\ a1 -> case  a0 of{
181                 ((Value27 int281)) -> case  a1 of{
182                   ((Value27 int282)) ->
183                     ( (case econt275 of
184                       ((F80 funDomains286)) -> funDomains286)
185                         ( (Value27 ( ( ( int281 ) + ( int282 ))))));
186                       _ -> err } ;
187                  _ -> err } ) )  ) ) ) ) )  ) ) )
188                 of ((F79 funDomains287)) ->
189                 funDomains287) state277 ))));
190
191      (Exp32 exp133 exp134 ) ->
192        ( (\econt288 ->
193          (F79(\state290 ->
194            (case ( ( ( de exp133 )
195              ( (F80 ( (\ a0 ->
```

```
196               ( ( de exp134 ) ( (F80 ( (\ a1 ->
197               if ( ( ( ( case  a0 of{
198                 ((Value27 int294)) -> True;
199                  _ -> False } ) ) && ( ( case  a1 of{
200                    ((Value27 int295)) -> True;
201                     _ -> False } ) ) ) )
202                  then ( ( (case econt288 of
203                    ((F80 funDomains299)) -> funDomains299)
204                      ( (Value29 ( ( ( gt a0 ) a1 ) )  ) ) ) ) )
205                  else ( err ) ) )  ) ) ) ) ) )  ) ) ) )
206             of ((F79 funDomains300)) -> funDomains300) state290 ))));
207
208       (Exp33 exp136 exp137 ) ->
209         ( (\econt301 ->
210           (F79(\state303 ->
211             (case ( ( ( de exp136 )
212               ( (F80 ( (\ a0 ->
213                 ( ( de exp137 ) ( (F80 ( (\ a1 ->
214                 if ( ( ( ( case  a0 of{
215                   ((Value27 int307)) -> True;
216                   _ -> False } ) ) && ( ( case  a1 of{
217                     ((Value27 int308)) -> True;
218                     _ -> False } ) ) ) )
219                   then ( ( (case econt301 of
220                     ((F80 funDomains312)) -> funDomains312)
221                       ( (Value29 ( ( ( lt a0 ) a1 ) )  ) ) ) )
222                   else ( err ) ) )  ) ) ) ) )  ) ) ) )
223             of ((F79 funDomains313)) -> funDomains313) state303 ))));
224
225       (Exp34 exp139 exp140 ) ->
226         ( (\econt314 ->
227           (F79(\state316 ->
228             (case ( ( de exp139 )
229               ( (F80 ( (\ a0 ->
230                 ( ( de exp140 ) ( (F80 ( (\ a1 ->
231                   case  a0 of{
232                     ((Value27 int320)) -> case  a1 of{
233                       ((Value27 int321)) ->
234                         ( (case econt314 of ((F80 funDomains325)) ->
235                           funDomains325)
```

```
236                          ( (Value27 ( ( ( int320 ) - ( int321 ))))) ;
237                        _ -> err } ;
238                      _ -> err } ) )  ) ) ) ) ) )  ) ) )
239              of ((F79 funDomains326)) -> funDomains326) state316 ))));
240
241      (Exp35 exp142 exp143 ) ->
242        ( (\econt327 ->
243          (F79(\state329 ->
244            (case ( ( de exp142 )
245              ( (F80 ( (\ a0 ->
246                ( ( de exp143 ) ( (F80 ( (\ a1 ->
247                  case  a0 of{
248                    ((Value27 int333)) -> case  a1 of{
249                      ((Value27 int334)) ->
250                        ( (case econt327 of ((F80 funDomains338)) ->
251                          funDomains338)
252                          ( (Value27 ( ( mod  ( int333 ) ( int334 ))))) ;
253                        _ -> err } ;
254                      _ -> err } ) )  ) ) ) ) )  ) ) )
255            of ((F79 funDomains339)) -> funDomains339) state329 ))))
256 }))
257
258 dc :: (Com -> (Cont -> Cont))
259 dc =
260   ( (\com340-> case com340 of {
261     (Com41 ide154 exp155 ) ->
262       ( (\cont341 ->( ( de exp155 )
263         ( (F80 ( (\ a0 ->
264           ( (F79 (\ ((T71__State (x343,x344,x345))) ->
265             ( (case cont341 of ((F79 funDomains347)) ->
266               funDomains347) ( (T71__State ( ( (F74 (update
267                 (case x343 of ((F74 funDomains348)) -> funDomains348)
268                 [ ide154 ]
269                 [ ( (U72 a0  ) ) ])  ) ) , x344 , x345 )  ) ) ) )  ) ) ) )
    ) ) ) ));
270
271     (Com42 exp157 ) ->
272       ( (\cont349 ->
273         ( ( de exp157 )
274           ( (F80 ( (\ a0 ->
```

```
275             ( (F79 (\ ((T71__State (x351,x352,x353))) ->
276               ( (case cont349 of ((F79 funDomains355)) -> funDomains355)
277               ( (T71__State ( x351 ,
278                               x352 ,
279                               ( (L76
280                                 (a0 :
281                                 (case x353 of ((L76 value356)) -> value356))))
282                )  ) ) ) )  ) ) ) )  ) ) ) ));
283
284     (Com43 exp160 com161 com162 ) ->
285       ( (\cont357 ->
286         ( ( de exp160 ) ( (F80 ( (\ a0 -> case  a0 of{
287           ((Value29 bool360)) ->
288             if ( bool360 )
289               then ( ( ( dc com161 ) cont357 ) )
290               else ( ( ( dc com162 ) cont357 ) ) ;
291           _ -> err } ) )  ) ) ) ));
292
293     (Com44 exp167 com168 ) ->
294       ( (\cont364 ->( ( de exp167 )
295         ( (F80 ( (\ a0 -> case  a0 of{
296           ((Value29 bool367)) ->
297             if ( bool367 )
298               then ( ( ( dc com168 )
299                 ( ( dc ( (Com44 exp167 com168  ) ) ) cont364 ) ) )
300               else ( cont364 ) ;
301           _ -> err } ) )  ) ) ) ));
302
303     (Com40 com172 com173 ) ->
304       ( (\cont371 ->( ( dc com172 ) ( ( ( dc com173 ) cont371 ) ) ) ));
305
306     (Com45 com175 exp176 ) ->
307       ( (\cont375 ->
308         ( ( dc com175 )
309           ( ( ( de exp176 ) ( (F80 ( (\ a0 -> case  a0 of{
310             ((Value29 bool379)) ->
311               if ( bool379 )
312                 then ( ( ( dc ( (Com45 com175 exp176  ) ) ) cont375 ) )
313                 else ( cont375 ) ;
314             _ -> err } ) )  ) ) ) ) ) ))
```

```
315 }))
316
317 gt :: (Value -> (Value -> Bool))
318 gt = ( (\value382-> case value382 of {
319 (Value27 int178) -> ( (\value383-> case value383 of {
320 (Value27 int179) ->( ( ( int178 ) > ( int179 ) ) ) ;
321 (_) ->False
322 }));
323 (_) ->( (\(Value27 int181) ->False ))
324 }))
325
326 lt :: (Value -> (Value -> Bool))
327 lt = ( (\value386-> case value386 of {
328 (Value27 int182) -> ( (\value387-> case value387 of {
329 (Value27 int183) ->( ( ( int182 ) < ( int183 ) ) ) ;
330 (_) ->False
331 }));
332 (_) ->( (\(Value27 int185) ->False ))
333 }))
334
335 result :: (Value -> (State -> ValueStatePair))
336 result =  ((\ a0 -> (\ a1 -> ( (T83__ValueStateTuple ( a0 , a1 )  ) ) )
    ) )
337
338 donothing :: (State -> Ans)
339 donothing =  ((\ a0 -> ( (U77 a0  ) ) ) )
340
341 checkNum :: (Value -> (State -> ValueStatePair))
342 checkNum =
343   (\ a0 -> ( (\ a1 -> (
344     if ( case  a0 of{
345       value394 -> True;
346       _ -> False } )
347       then ( ( (T83__ValueStateTuple ( a0 , a1 )  ) ) )
348       else ( ( (U84 (E81 (Enum0__Error  ) ) ) ) )  ) )  ) )
349
350 checkBool :: (Value -> (State -> ValueStatePair))
351 checkBool =
352   (\ a0 -> ( (\ a1 -> (
353     if ( case  a0 of{
```

```
354       ((Value29 bool396)) -> True;
355       _ -> False } )
356       then ( ( (T83__ValueStateTuple ( a0 , a1 )  ) ) )
357       else ( ( (U84 (E81 (Enum0__Error  ) ) ) ) )  ) )  ) )
358
359 equals :: (Value -> (Value -> Bool))
360 equals = ( (\value397-> case value397 of {
361 (Value29 bool186) -> ( (\value398-> case value398 of {
362 (Value29 bool187) ->( ( ( bool186 ) == ( bool187 ) ) ) ;
363 (_) ->False
364 }));
365 (Value27 int188) -> ( (\value399-> case value399 of {
366 (Value27 int189) ->( ( ( int188 ) == ( int189 ) ) ) ;
367 (_) ->False
368 }))
369 }))
370
371 err :: Cont
372 err = (F79(\state400-> ((U78(E81(Enum0__Error ))))))
373
374
375
376
377 main = do
378   (x:xs) <- getArgs
379   progL <- readFile x
380   inL <- sReaderNotus (head(matchInFiles [(Stdin)] xs))
381   sWriterNotus (head(matchOutFiles [(Stdout)] xs))
382     (dprog  (parse (alexScanTokens progL)) inL)
```

Listagem C.9: Módulo principal do interpretador *Tiny*

Apêndice D

A Linguagem Small

D.1 Definição da Linguagem

Esta seção apresenta os módulos que compõem a especificação de *Small*. Um programa em *Small* inicia-se com a palavra `program` seguida por pelo menos um comando, que pode conter declarações, expressões e outros comandos. Os comandos existentes em *Small* são os de atribuição, condicional, repetição, saída, chamada de procedimento e bloco contendo declaração e comandos. As expressões em *Small* são números, booleanos, aritméticas, relacionais, condicional, leitura e chamada de função. Em *Small* pode-se declarar constantes, variáveis, procedimentos e funções. O conjunto de operadores binários de *Small* é $\{+, -, *, /, <, >, ==, / =\}$

D.1.1 Kernel

O módulo `Kernel`, o mesmo usado nas definições de *Nano* e *Tiny* mostrado na Listagem B.4, define que espaços em branco e comentários são ignorados pelo analisador léxico.

D.1.2 Declarações

O módulo de declarações de *Small* da Listagem D.1 define a sintaxe e a semântica das declarações de *Small*, como palavras-chave, gramática concreta e as equações semânticas que descrevem a denotação dessas declarações.

```
1 module Declarations
2   import Expressions, Domains, Util;
3
4   // Syntax
5   d : Dec ::= d ";" simpleDec | simpleDec : simpleDec;
6   simpleDec : Dec ::= "const" Syntax.id "=" e
```

```
7                       | "var" Tokens.id "=" e
8                       | "proc" Tokens.id "(" Tokens.id1 ")" c "end"
9                       | "fun" Tokens.id "(" Tokens.id1 ")" e "end";
10
11    // Semantics
12    ddec ["const" id "=" e] env dc =
13      dr e env (\ev -> case ev of
14        { dv -> dc newEnv[id <- dv];
15          _  -> err});
16
17    ddec ["var" id "=" e] env dc =
18      evaluate {dr e env; ref (\loc -> dc newEnv[id <- loc])};
19
20    ddec ["proc" id "(" id1 ")" c "end"] env dc =
21        let {proc = (\cc ev ->
22          ref (\loc -> dcom c env[id1 <- loc,id <- proc] cc) ev)}
23        in dc newEnv[id <- proc];
24
25    ddec ["fun" id "(" id1 ")" e "end"] env dc =
26      let {fun = (\ec ev ->
27        ref (\loc -> dexp e env[id1 <- loc,id <- fun] ec) ev)}
28      in dc newEnv[id <- fun];
29
30    ddec [d1 ";" d2] env dc =
31      ddec d1 env (\env1 -> ddec d2 env3 (\env2 -> dc env4
32        where {env4 =
33          (\ide1 -> if env2 ide1 is unbound
34                    then env1 ide1
35                    else env2 ide1)})
36        where {env3 = (\ide2 -> if env1 ide2 is unbound
37                      then env ide2
38                      else env1 ide2)});
39  end
```

Listagem D.1: Módulo `Domains` da definição de *Small*

D.1.3 Domínios

O módulo de domínios de *Small* da Listagem D.2 define os domínios sintáticos e semânticos da especificação denotacional padrão de *Small*.

```
1  module Domains
2    // Syntactic domains
3    public syntactic Opr, Id;
4
5    // Semantic domains
6    Bas   = Int;
7    Loc   = String;
8    Bv    = Bas;
9    Dv    = Loc | Rv | Proc | Fun;
10   Sv    = File | Rv;
11   Ev    = Dv;
12   Rv    = Bool | Bv;
13   File  = Rv*;
14   Env   = Id -> EnvValue;
15   Store = Loc -> StoreValue;
16   Cc    = Store -> Ans;
17   Ec    = Ev -> Cc;
18   Dc    = Env -> Cc;
19   Proc  = Cc -> Ec;
20   Fun   = Ec -> Ec;
21   Ans   = AnsValue (Rv, Ans) | {error, stop} ;
22   Unbound = {unbound};
23   Unused = {unused};
24   EnvValue = Dv | Unbound;
25   StoreValue = Sv | Unused;
26 end
```

Listagem D.2: Módulo `Domains` da definição de *Small*

D.1.4 Funções Semânticas

O módulo das funções semânticas de *Small* da Listagem D.3 define as assinaturas das equações de denotação dos construtos de *Small*. A sintaxe e a semântica de um programa em *Small* também é definida nesse módulo.

D.1.5 Terminais

O módulo dos terminais de *Small* da Listagem D.4 define os *tokens* de *Small*.

```
module Semantics
  import Tokens, Expressions, Commands, Declarations, Util;

  // Semantic functions
  function dopr  : Opr -> Rv -> Rv -> Ec -> Cc;
  function dprog : Pro -> File -> Ans;
  function dr    : Exp -> Env -> Ec -> Cc;
  function dexp  : Exp -> Env -> Ec -> Cc;
  function dcom  : Com -> Env -> Cc -> Cc;
  function ddec  : Dec -> Env -> Dc -> Cc;

  // Program syntax
  p : Pro ::= "program" c;

  // Program semantics
  dprog ["program" c] i =
    dcom c newEnv (\store -> stop) initialStore["input" <- i];

  function initialStore : Loc -> StoreValue;
  initialStore loc = unused;
end
```

Listagem D.3: Módulo `Semantics` da definição de *Small*

```
module Tokens
  import Domains, Expressions, Commands;

  token id   : Id  = [a-zA-Z_][a-zA-Z0-9_]*;
  token bas  : Bas  = [0-9]+ is asInteger;
  token bool : Bool = "True" | "False" is asBoolean;

end
```

Listagem D.4: Módulo `Tokens` da definição de *Small*

D.1.6 Expressões

O módulo de expressões de *Small* da Listagem D.5 define a sintaxe e a semântica de expressões, como palavras-chave, gramática concreta e as equações semânticas que descrevem a denotação dessas expressões.

```
1  module Expressions
2    import Tokens, Domains, Util, Operators;
3
4    // Syntax
5    public e : Exp ::= relexp : relexp;
6    relexp:Exp ::= relexp relop addexp
7                 | addexp : addexp;
8    addexp:Exp ::= addexp addop multexp
9                 | multexp : multexp;
10   multexp:Exp ::= multexp multop unary
11                | unary : unary;
12   unary:Exp ::= primaryexp:primaryexp;
13   primaryexp : Exp ::= bas:bas
14                     | bool:bool
15                     | id
16                     | "read"
17                     | "(" e ")":e
18                     | compexp:compexp;
19   compexp:Exp ::= id "(" e2 ")" : ["funCall" id e2]
20                     | "if" e "then" e1 "else" e2 "end";
21
22   // Semantics
23   dr exp env ec = evaluate{dexp exp env; deref; isRv; ec};
24   dexp int env ec = ec int;
25   dexp bool env ec = ec bool;
26   dexp ["read"] env ec store = case store "input" of {
27                     unused -> error;
28                     rv78*     -> let {rv1 = headRv rv78*;
29                                       rv2* = tailRv rv78* } in
30                       ec (rv1)(store["input" <- rv2*])};
31
32
33   dexp [id] env ec = case env id of
34               {dv -> ec dv;
35                _  -> err};
36
37   dexp ["funCall" id e2] env ec = evaluate {
38                     dexp [id] env;
39                     isFun (\fun -> evaluate {dr e2 env; fun; ec})};
40
41   dexp ["if" e "then" e1 "else" e2 "end"] env ec = evaluate {
42                            dr e env;
43                            isBool (\bool -> if bool
44                                       then dexp e1 env ec
45                                       else dexp e2 env ec)};
46
47   dexp [e1 opr e2] env ec = dr e1 env ec1
48     where {ec1 = \ev1 -> dr e2 env ec2
49           where {ec2 = \ev2 -> case ev1 of {
50                       rv1 -> case ev2 of {
51                         rv2 -> dopr opr rv1 rv2 ec}}}};
52 end
```

Listagem D.5: Módulo `Expressions` da definição de *Small*

D.1.7 Comandos

O módulo de comandos de *Small* da Listagem D.6 define a sintaxe e a semântica de comandos, como as palavras-chave, gramáticas concreta e abstrata e as equações semânticas que descrevem a denotação desses comandos.

```
1  module Commands
2    import Tokens, Domains, Semantics, Expressions, Declarations, Util;
3
4    // Syntax
5    c : Com ::= c ";" simpleCom
6              | simpleCom : simpleCom;
7    simpleCom : Com ::= e1 ":=" e2
8                      | "output" e
9                      | "call" id "(" e2 ")": ["procCall" id e2]
10                     | "if" e "then" c1 "else" c2 "end"
11                     | "while" e "do" c "end"
12                     | "begin" d ";" c "end";
13
14   // Semantics
15   dcom [e1 ":=" e2] env cc = evaluate {dexp e1 env; isLoc ec}
16     where {ec = (\ev ->
17              case ev of
18                { loc -> cc1
19                  where {cc1 = evaluate {dr e2 env; update loc; cc}}
20                })
21            };
22
23   dcom ["output" e] env cc =
24     dr e env (\ev store -> case ev of {
25       rv -> AnsValue(rv,cc store);
26       _  -> error});
27
28   dcom ["procCall" id e2] env cc =
29     evaluate {dexp [id] env;
30               isProc (\proc -> evaluate {dr e2 env; proc; cc})};
31
32   dcom ["if" e "then" c1 "else" c2 "end"] env cc =
33     evaluate {dr e env; isBool (\bool -> if bool
34                                          then dcom c1 env cc
35                                          else dcom c2 env cc)};
36
37   dcom ["while" e "do" c "end"] env cc =
38     evaluate {dr e env; isBool (\bool ->
39       if bool
40         then dcom c env (dcom ["while" e "do" c "end"] env cc)
41         else cc)};
42
43   dcom ["begin" d ";" c "end"] env cc = ddec d env dc
44     where {dc = \env1 -> dcom c env2 cc
45         where {env2 = \ide -> if env1 ide is unbound
46                               then env ide
47                               else env1 ide}};
48
49   dcom [c1 ";" c2] env cc =
50     evaluate {dcom c1 env; dcom c2 env; cc};
51 end
```

Listagem D.6: Módulo *Commands* da definição de *Small*

D.1.8 Operadores

O módulo de operadores de *Small* da Listagem D.7 define a sintaxe e a semântica de operadores, como palavras-chave, gramática concreta e as equações semânticas que descrevem a denotação desses operadores.

```
1  module Operators
2    import Tokens, Domains;
3
4    // Syntax
5    public addop  : Opr ::= "+" | "-" ;
6    public multop : Opr ::= "*" | "/" ;
7    public relop  : Opr ::= "<" | ">" | "==" | "/=";
8
9    // Semantics
10   dopr opr rv1 rv2 ec = let { ev = case rv1 of {
11                       int1 -> case rv2 of {
12                             int2 ->  case opr of {
13                                    ["+"]  -> int1 + int2;
14                                    ["-"]  -> int1 - int2;
15                                    ["*"]  -> int1 * int2;
16                                    ["/"]  -> int1 / int2;
17                                    ["<"]  -> int1 < int2;
18                                    [">"]  -> int1 > int2;
19                                    ["=="] -> int1 = int2;
20                                    ["/="] -> int1 != int2
21                                  }
22                          }
23                       }
24               } in ec ev;
25 end
```

Listagem D.7: Módulo `Operators` da definição de *Small*

D.1.9 Funções Auxiliares

O módulo de funções auxiliares de *Small* da Listagem D.8 define funções auxiliares utilizadas pelas equações de denotação dos construtos de *Small*.

```
1  module Util
2    import Domains;
3
```

```
4   function cont : Ec -> Ec;
5   cont ec ev store = case ev of {
6                        loc -> case (store loc) of {
7                                 rv -> ec rv store;
8                                 unused -> error;
9                                 _ -> error};
10                       _   -> error};
11
12  function update : Loc -> Cc -> Ec;
13  update loc cc ev store = case ev of
14                {rv -> cc store[loc <- rv];
15                 _  -> error};
16
17    function deref : Ec -> Ec;
18    deref ec ev store = case ev of {
19                    loc -> cont ec loc store;
20                    _   -> ec ev store};
21
22    function new : Store -> Loc;
23    new store = findNew store "";
24
25    function findNew : Store -> Loc -> Loc;
26    findNew store string =
27      if (store string) is unused
28        then string
29        else findNew store newLoc
30      where {newLoc = string ++  "x"};
31
32    function ref : Ec -> Ec;
33    ref ec ev store = case (new store) of {
34              loc   -> update loc (ec loc) ev store};
35
36    function err : Cc;
37    err store = error;
38
39    function newEnv : Env;
40    // nao reconheceu que (\id -> unbound) eh Env
41    newEnv = \ide -> unbound;
42
43    function headRv : Rv* -> Rv;
```

```
44      headRv rv:rvs* = rv;
45
46      function tailRv : Rv* -> Rv*;
47      tailRv rv:rvs* = rvs*;
48
49      function isRv : Ec -> Ec;
50      isRv econt ev = if ev is rv then econt ev else err;
51
52      function isFun : Ec -> Ec;
53      isFun econt ev = if ev is fun then econt ev else err;
54
55      function isBool : Ec -> Ec;
56      isBool econt ev = if ev is bool then econt ev else err;
57
58      function isLoc : Ec -> Ec;
59      isLoc econt ev = if ev is loc then econt ev else err;
60
61      function isProc : Ec -> Ec;
62      isProc econt ev = if ev is proc then econt ev else err;
63  end
```

Listagem D.8: Módulo `Util` da definição de *Small*

D.1.10 Módulo Principal

O módulo principal na Listagem D.9 define a função de avaliação de *AST* responsável por iniciar a execução de programas em *Small* por meio da cláusula `semantics`, além de funções auxiliares para tratamento da entrada e saída dos programas em *Small*. A cláusula `syntax` define que a variável de gramática `p` é o símbolo inicial da gramática concreta de *Small*. A cláusula `input` define que os dados de entrada serão lidos na entrada padrão, e a cláusula `output` define que os dados de saída serão escritos na saída padrão.

```
1  module Main
2    import Domains, Tokens, Semantics, Kernel;
3    syntax p;
4    semantics dmain;
5
6    input stdin;
7    output stdout;
8
9    function dmain : Pro -> String -> String;
10   dmain p inp = evaluate{showOutput; dprog p; evalInput inp};
11
12   function evalInput : String -> Rv*;
13   evalInput s = (intToRv(nextInt s)):evalInput (ignoreInt s);
14
15   function intToRv : Int -> Rv;
16   intToRv i = i;
17
18   function showOutput : Ans -> String;
19   showOutput stop = "";
20   showOutput error = "program error";
21   showOutput AnsValue (rv,ans) =
22     (case rv of { int  -> integerToString int;
23                   bool -> booleanToString bool })
24      ++ "\t" ++ showOutput ans;
25 end
```

Listagem D.9: Módulo *Principal* da definição de *Small*

D.2 Interpretador de *Small*

O interpretador de *Small* compreende os analisadores léxico e sintático, e o programa principal. As listagens contendo os arquivos gerados pelo compilador de *Notus* para a especificação de *Small* são mostradas a seguir.

D.2.1 Analisador Léxico.

O analisador léxico de *Small* é produzido pelo gerador de analisadores léxicos *Alex* a partir do arquivo de especificação léxica *Lexer.x* gerado pelo compilador de *Notus*. O arquivo *Lexer.x* é mostrado na Listagem D.10.

```
1  {
2  module Lexer where
3  import NotusDefault
4  import DataModule
5  import NotusFunctions
6  }
7
8  %wrapper "basic"
9
10 @kernel__layoutchar = " "|""[\n]""|""[\t]""|""[\r]""|""[\f]""
11 @kernel__comment = "//".*
12
13 tokens :-
14
15 ":="      { \s->(T__commands__token__15)}
16 "+"     { \s->(T__operators__token__44)}
17 "/"     { \s->(T__operators__token__47)}
18 "<"     { \s->(T__operators__token__48)}
19 "True"|"False"      { \s->(T__tokens__bool( (asBoolean s)))}
20 ")"     { \s->(T__expressions__token__6)}
21 "*"     { \s->(T__operators__token__46)}
22 ";"     { \s->(T__commands__token__14)}
23 "="     { \s->(T__declarations__token__33)}
24 "=="      { \s->(T__operators__token__50)}
25 [a-zA-Z_][a-zA-Z0-9_]*
26   { \s->lookupT tokens__idTree s (T__tokens__id(Id__52 ( s)))}
27 "("     { \s->(T__expressions__token__5)}
28 "/="      { \s->(T__operators__token__51)}
29 [0-9]+      { \s->(T__tokens__bas(Bas__53 (asInteger s)))}
```

```
30 ">"    { \s->(T__operators__token__49)}
31 "-"    { \s->(T__operators__token__45)}
32
33 @kernel__layoutchar|@kernel__comment     ;
34
35 {
36 -- The token type:
37
38 data Token__Main = T__semantics__token__3
39                  | T__expressions__token__4
40                  | T__expressions__token__5
41                  | T__expressions__token__6
42                  | T__expressions__token__10
43                  | T__expressions__token__11
44                  | T__expressions__token__12
45                  | T__expressions__token__13
46                  | T__commands__token__14
47                  | T__commands__token__15
48                  | T__commands__token__16
49                  | T__commands__token__17
50                  | T__commands__token__25
51                  | T__commands__token__26
52                  | T__commands__token__28
53                  | T__declarations__token__32
54                  | T__declarations__token__33
55                  | T__declarations__token__34
56                  | T__declarations__token__36
57                  | T__declarations__token__40
58                  | T__operators__token__44
59                  | T__operators__token__45
60                  | T__operators__token__46
61                  | T__operators__token__47
62                  | T__operators__token__48
63                  | T__operators__token__49
64                  | T__operators__token__50
65                  | T__operators__token__51
66                  | T__tokens__id Id__Domains
67                  | T__tokens__bas Bas__Tokens
68                  | T__tokens__bool Bool__NotusDefault
69   deriving Eq
```

```
70
71
72
73
74 data Tree a b = Leaf (a,b)
75          | Node (a,b) (Tree a b) (Tree a b)
76          | Null
77      deriving Eq
78
79 lookupT :: Ord a => Tree a Token__Main -> a -> Token__Main -> Token__Main
80 lookupT (Leaf (a,b)) s tReturn
81   | s == a = b
82   | otherwise = tReturn
83 lookupT (Node (n,t) l r) s tReturn
84   | n == s  = t
85   | s > n   = lookupT r s tReturn
86   | s < n   = lookupT l s tReturn
87 lookupT Null _ tReturn = tReturn
88
89 tokens__idTree =
90   (Node ("if",(T__expressions__token__10))
91   (Node ("do",(T__commands__token__26))
92   (Node ("call",(T__commands__token__17))
93     (Leaf ("begin",(T__commands__token__28)))
94     (Leaf ("const",(T__declarations__token__32))))
95   (Node ("end",(T__expressions__token__13))
96     (Leaf ("else",(T__expressions__token__12)))
97     (Leaf ("fun",(T__declarations__token__40)))))
98   (Node ("read",(T__expressions__token__4))
99   (Node ("proc",(T__declarations__token__36))
100    (Leaf ("output",(T__commands__token__16)))
101    (Leaf ("program",(T__semantics__token__3))))
102  (Node ("var",(T__declarations__token__34))
103    (Leaf ("then",(T__expressions__token__11)))
104    (Leaf ("while",(T__commands__token__25))))))
105
106
107 }
```

Listagem D.10: Arquivo de especificação léxica *Lexer.x* para o *Alex*

D.2.2 Analisador Sintático

O analisador sintático é produzido pelo gerador de analisadores sintáticos *Happy* a partir do arquivo de especificação sintática *Syntactic.y* gerado pelo compilador de *Notus*. O arquivo *Syntactic.y* é mostrado na Listagem D.11.

```
1  {
2  module Syntactic where
3  import Char
4  import Lexer
5  import NotusDefault
6  import DataModule
7  import NotusFunctions
8  }
9
10 %name parse semantics_p
11 %tokentype { Token_Main }
12
13 %token
14   tokens_id       { T_tokens_id $$ }
15   tokens_bas      { T_tokens_bas $$ }
16   tokens_bool       { T_tokens_bool $$ }
17   "program"      { T_semantics_token_3 }
18   "read"       { T_expressions_token_4 }
19   "("      { T_expressions_token_5 }
20   ")"      { T_expressions_token_6 }
21   "if"       { T_expressions_token_10 }
22   "then"       { T_expressions_token_11 }
23   "else"       { T_expressions_token_12 }
24   "end"      { T_expressions_token_13 }
25   ";"      { T_commands_token_14 }
26   ":="       { T_commands_token_15 }
27   "output"       { T_commands_token_16 }
28   "call"       { T_commands_token_17 }
29   "while"      { T_commands_token_25 }
30   "do"       { T_commands_token_26 }
31   "begin"      { T_commands_token_28 }
32   "const"      { T_declarations_token_32 }
33   "="      { T_declarations_token_33 }
34   "var"      { T_declarations_token_34 }
35   "proc"       { T_declarations_token_36 }
```

```
36   "fun"      { T__declarations__token__40 }
37   "+"      { T__operators__token__44 }
38   "-"      { T__operators__token__45 }
39   "*"      { T__operators__token__46 }
40   "/"      { T__operators__token__47 }
41   "<"      { T__operators__token__48 }
42   ">"      { T__operators__token__49 }
43   "=="       { T__operators__token__50 }
44   "/="       { T__operators__token__51 }
45 %%
46 semantics__p     : "program" commands__c      { Pro__57 $2 }
47 expressions__e     : expressions__relexp      {  $1 }
48 expressions__relexp     :
49   expressions__relexp operators__relop expressions__addexp  { Exp__60 $1 $2 $3 }
50 | expressions__addexp      {  $1 }
51 expressions__addexp     :
52   expressions__addexp operators__addop expressions__multexp {Exp__60 $1 $2 $3}
53 | expressions__multexp     {  $1 }
54 expressions__multexp     :
55   expressions__multexp operators__multop expressions__unary { Exp__60 $1 $2 $3 }
56 | expressions__unary     {  $1 }
57 expressions__unary     :
58   expressions__primaryexp      {  $1 }
59 expressions__primaryexp     :
60   tokens__bas { ignoreIndirectionFun__97 $1  }
61 | tokens__bool { ignoreIndirectionFun__98 $1  }
62 | tokens__id     { Exp__61 $1 }
63 | "read"     { Exp__62 }
64 | "(" expressions__e ")"     {  $2 }
65 | expressions__compexp     {  $1 }
66 expressions__compexp     :
67   tokens__id "(" expressions__e ")"      { Exp__63 $1 $3 }
68 | "if" expressions__e "then" expressions__e "else" expressions__e "end"
69   { Exp__64 $2 $4 $6 }
70 commands__c     :
71   commands__c ";" commands__simpleCom      { Com__65 $1 $3 }
72 | commands__simpleCom      {  $1 }
73 commands__simpleCom     :
74   expressions__e ":=" expressions__e     { Com__66 $1 $3 }
75 | "output" expressions__e      { Com__67 $2 }
```

```
76  | "call" tokens__id "(" expressions__e ")"    { Com__68 $2 $4 }
77  | "if" expressions__e "then" commands__c "else" commands__c "end"
78    { Com__69 $2 $4 $6 }
79  | "while" expressions__e "do" commands__c "end"     { Com__70 $2 $4 }
80  | "begin" declarations__d ";" commands__c "end"     { Com__71 $2 $4 }
81  declarations__d    :
82    declarations__d ";" declarations__simpleDec     { Dec__72 $1 $3 }
83  | declarations__simpleDec     {  $1 }
84  declarations__simpleDec    :
85    "const" tokens__id "=" expressions__e     { Dec__73 $2 $4 }
86  | "var" tokens__id "=" expressions__e     { Dec__74 $2 $4 }
87  | "proc" tokens__id "(" tokens__id ")" commands__c "end" { Dec__75 $2 $4 $6 }
88  | "fun" tokens__id "(" tokens__id ")" expressions__e "end" { Dec__76 $2 $4 $6
89  operators__addop    : "+"     { Opr__77 }
90                      | "-"     { Opr__78 }
91  operators__multop    : "*"    { Opr__79 }
92                       | "/"    { Opr__80 }
93  operators__relop    : "<"     { Opr__81 }
94                      | ">"     { Opr__82 }
95                      | "=="    { Opr__83 }
96                      | "/="    { Opr__84 }
97
98  {
99
100 happyError :: [Token__Main] -> a
101 happyError _ = error "Parse error"
102
103
104 ----------------------Ignore Indirections Functions----------------------
105
106 ignoreIndirectionFun__97 :: Bas__Tokens -> Exp__Expressions
107 ignoreIndirectionFun__97 igInd=
108   case igInd of
109     (Bas__53 a1) -> (Exp__85 a1)
110
111 ignoreIndirectionFun__98 :: Bool__NotusDefault -> Exp__Expressions
112 ignoreIndirectionFun__98 igInd=
113   case igInd of
114     ( a1) -> (Exp__86 a1)
115
```

```
116 }
```

Listagem D.11: Arquivo de especificação sintática *Syntactic.y* para o *Happy*

D.2.3 Gramática Abstrata

A Listagem D.12 mostra o módulo `DataModule`, gerado pelo compilador, com os *datatypes* que representam a gramática abstrata de *Small*.

```
1  module DataModule where
2
3
4  data Id__Domains = Id__52 String
5    deriving Eq
6
7  data Bas__Tokens = Bas__53 Int
8    deriving Eq
9
10 data Pro__Semantics = Pro__57 Com__Commands
11   deriving Eq
12
13 data Exp__Expressions =
14   Exp__58 Exp__Expressions  Opr__Domains  Exp__Expressions
15 | Exp__59 Exp__Expressions  Opr__Domains  Exp__Expressions
16 | Exp__60 Exp__Expressions  Opr__Domains  Exp__Expressions
17 | Exp__61 Id__Domains
18 | Exp__62
19 | Exp__63 Id__Domains  Exp__Expressions
20 | Exp__64 Exp__Expressions  Exp__Expressions  Exp__Expressions
21 | Exp__85 Int
22 | Exp__86 Bool
23   deriving Eq
24
25 data Com__Commands =
26   Com__65 Com__Commands  Com__Commands
27 | Com__66 Exp__Expressions  Exp__Expressions
28 | Com__67 Exp__Expressions
29 | Com__68 Id__Domains  Exp__Expressions
30 | Com__69 Exp__Expressions  Com__Commands  Com__Commands
31 | Com__70 Exp__Expressions  Com__Commands
32 | Com__71 Dec__Declarations  Com__Commands
33   deriving Eq
34
35 data Dec__Declarations =
36   Dec__72 Dec__Declarations  Dec__Declarations
37 | Dec__73 Id__Domains  Exp__Expressions
38 | Dec__74 Id__Domains  Exp__Expressions
39 | Dec__75 Id__Domains  Id__Domains  Com__Commands
40 | Dec__76 Id__Domains  Id__Domains  Exp__Expressions
41   deriving Eq
42
43 data Opr__Domains = Opr__77
44                   | Opr__78
45                   | Opr__79
46                   | Opr__80
47                   | Opr__81
48                   | Opr__82
49                   | Opr__83
50                   | Opr__84
51   deriving Eq
```

Listagem D.12: Módulo `DataModule` contendo os *datatypes* da gramática abstrata de *Small*

D.2.4 Programa Principal

O programa principal é composto pelas funções semânticas da especificação de *Small*, e pela função principal main que inicia a execução do interpretador. A Listagem D.13 mostra o programa principal gerado para a especificação de *Small*.

```
1  module Main where
2
3  import Lexer
4  import Syntactic
5  import DataModule
6  import NotusDefault
7  import NotusFunctions
8  import Data.Bits
9  import System
10
11 data Bas__Domains = D__99__NotusDefault Int__NotusDefault
12   deriving Eq
13
14
15 data Loc__Domains = D__100__NotusDefault String__NotusDefault
16   deriving Eq
17
18
19 data Bv__Domains = D__101__Domains Bas__Domains
20   deriving Eq
21
22
23 data Dv__Domains = U__102__Domains Loc__Domains
24                  | U__103__Domains Rv__Domains
25                  | U__104__Domains Proc__Domains
26                  | U__105__Domains Fun__Domains
27
28
29 data Sv__Domains = U__106__Domains File__Domains
30                  | U__107__Domains Rv__Domains
31   deriving Eq
32
33
34 data Ev__Domains = D__108__Domains Dv__Domains
35
```

```
36
37 data Rv__Domains = U__109__Domains Bool__NotusDefault
38                  | U__110__Domains Bv__Domains
39   deriving Eq
40
41
42 data File__Domains = L__111__Domains [ Rv__Domains ]
43   deriving Eq
44
45
46 data Env__Domains = F__112__Domains  (Id__Domains -> EnvValue__Domains)
47
48
49 data Store__Domains = F__113__Domains  (Loc__Domains -> StoreValue__Domains)
50
51
52 data Cc__Domains = F__114__Domains  (Store__Domains -> Ans__Domains)
53
54
55 data Ec__Domains = F__115__Domains  (Ev__Domains -> Cc__Domains)
56
57
58 data Dc__Domains = F__116__Domains  (Env__Domains -> Cc__Domains)
59
60
61 data Proc__Domains = F__117__Domains  (Cc__Domains -> Ec__Domains)
62
63
64 data Fun__Domains = F__118__Domains  (Ec__Domains -> Ec__Domains)
65
66
67 data Ans__Domains = T__119__domains__AnsValue (Rv__Domains , Ans__Domains)
68                  | E__120__Domains Enum__0__Domains
69   deriving Eq
70
71
72 data Unbound__Domains = E__121__Domains Enum__1__Domains
73   deriving Eq
74
75
```

```
76  data Unused__Domains = E__122__Domains Enum__2__Domains
77    deriving Eq
78
79
80  data EnvValue__Domains = U__123__Domains Dv__Domains
81                         | U__124__Domains Unbound__Domains
82
83
84  data StoreValue__Domains = U__125__Domains Sv__Domains
85                           | U__126__Domains Unused__Domains
86    deriving Eq
87
88
89  data Enum__0__Domains = Enum__0__Stop
90                        | Enum__0__Error
91    deriving Eq
92
93
94  data Enum__1__Domains = Enum__1__Unbound
95    deriving Eq
96
97
98  data Enum__2__Domains = Enum__2__Unused
99    deriving Eq
100
101
102 ---------- Functions ----------
103
104 main__dmain :: (Pro__Semantics -> (String__NotusDefault -> String__NotusDefault)
105 main__dmain =
106   ( (\pro__semantics__299 ->
107     ( (\string__notusdefault__300 ->
108       ( main__showOutput ( ( semantics__dprog pro__semantics__299 )
109         ( (L__111__Domains ( main__evalInput string__notusdefault__300 )))))))))
110
111 main__evalInput :: (String__NotusDefault -> [Rv__Domains])
112 main__evalInput =
113   ((\string__notusdefault__305 ->
114     ((( main__intToRv((nextInt string__notusdefault__305 )))):
115       ( main__evalInput ( ( ignoreInt string__notusdefault__305 ))))))
```

```
116
117 main__intToRv :: (Int__NotusDefault -> Rv__Domains)
118 main__intToRv =
119   ( (\int__notusdefault__310->
120     ((U__110__Domains(
121       D__101__Domains(D__99__NotusDefault (int__notusdefault__310 )))))))
122
123 main__showOutput :: (Ans__Domains -> String__NotusDefault)
124 main__showOutput = ( (\ans__domains__311-> case ans__domains__311 of {
125 ((E__120__Domains(Enum__0__Stop ))) ->"" ;
126 ((E__120__Domains(Enum__0__Error ))) ->"program␣error" ;
127 ((T__119__domains__AnsValue (( rv__domains__131),( ans__domains__132)))) ->
128   ( ( ( case  rv__domains__131 of{
129  ((U__110__Domains(D__101__Domains
130   (D__99__NotusDefault int__notusdefault__313)))) ->
131     ( integerToString int__notusdefault__313 ) ;
132  ((U__109__Domains bool__notusdefault__315)) ->
133   ( booleanToString bool__notusdefault__315 ) } )  ++
134     "\t" )  ++
135     ( main__showOutput ans__domains__132 ) )
136 }))
137
138 semantics__dopr :: (Opr__Domains -> (Rv__Domains -> (Rv__Domains ->
139   (Ec__Domains -> Cc__Domains))))
140 semantics__dopr =
141   ((\opr__domains__317->((\rv__domains__318 ->
142       ((\rv__domains__320 ->((\ec__domains__322 ->(let {
143           ev__domains__324 = case  rv__domains__318 of{
144             ((U__110__Domains(D__101__Domains
145               (D__99__NotusDefault int__notusdefault__325)))) ->
146                 case  rv__domains__320 of{
147                 ((U__110__Domains
148                   (D__101__Domains(D__99__NotusDefault
149                     int__notusdefault__326)))) -> case  opr__domains__317 of{
150                       ((Opr__77 )) ->
151                         ( (D__108__Domains(U__103__Domains
152                           (U__110__Domains (D__101__Domains
153                             (D__99__NotusDefault
154                             ((int__notusdefault__325 ) +
155                               ( int__notusdefault__326 ))))))));
```

```
156                    ((Opr_  _78 )) ->
157                      ( (D__108__Domains(U__103__Domains
158                        (U__110__Domains (D__101__Domains
159                          (D__99__NotusDefault
160                          ((int__notusdefault__325) -
161                            (int__notusdefault__326 ))))))));
162                    ((Opr__79 )) ->
163                      ( (D__108__Domains(U__103__Domains
164                      (U__110__Domains (D__101__Domains
165                        (D__99__NotusDefault
166                        ((int__notusdefault__325) *
167                          (int__notusdefault__326 ))))))));
168                    ((Opr__80 )) ->
169                      ( (D__108__Domains (U__103__Domains
170                      (U__110__Domains (D__101__Domains
171                        (D__99__NotusDefault
172                        (div (int__notusdefault__325)
173                          (int__notusdefault__326 ))))))));
174                    ((Opr__81 )) ->
175                      ( (D__108__Domains (U__103__Domains
176                      (U__109__Domains
177                        ((int__notusdefault__325) <
178                          (int__notusdefault__326 ))))));
179                    ((Opr__82 )) ->
180                      ( (D__108__Domains (U__103__Domains
181                      (U__109__Domains
182                        ((int__notusdefault__325) >
183                          (int__notusdefault__326 ))))));
184                    ((Opr__83 )) ->
185                      ( (D__108__Domains (U__103__Domains
186                      (U__109__Domains
187                        ((int__notusdefault__325) ==
188                          (int__notusdefault__326 ))))));
189                    ((Opr__84 )) ->
190                      ( (D__108__Domains (U__103__Domains
191                      (U__109__Domains
192                        ((int__notusdefault__325) /=
193                          (int__notusdefault__326 ))))))}}}}
194                      in  ( (case ec__domains__322 of
195                        ((F__115__Domains funDomains__327)) ->
```

```
196                                funDomains__327) ev__domains__324 ))))))))))
197
198 semantics__dprog :: (Pro__Semantics -> (File__Domains -> Ans__Domains))
199 semantics__dprog =
200   (\( (Pro__57 com__commands__137 ) ) ->( (\file__domains__329 ->
201     ( (case ( ( ( semantics__dcom com__commands__137 ) util__newEnv )
202       ( (F__114__Domains ( (\ store__domains__331 ->
203         ( (E__120__Domains (Enum__0__Stop))))))))
204     of ((F__114__Domains funDomains__335)) -> funDomains__335)
205       ((F__113__Domains (update semantics__initialStore
206         [((D__100__NotusDefault "input"  ) ) ]
207         [( (U__125__Domains (U__106__Domains file__domains__329)))]
208   )))))))
209
210 semantics__dr :: (Exp__Expressions -> (Env__Domains ->
211   (Ec__Domains -> Cc__Domains)))
212 semantics__dr =
213   ( (\exp__expressions__336-> ( (\env__domains__337 ->
214     ( (\ec__domains__339 ->
215       ( ( ( semantics__dexp exp__expressions__336 ) env__domains__337 )
216       ( util__deref ( util__isRv ec__domains__339 ))))))))
217
218 semantics__dexp :: (Exp__Expressions -> (Env__Domains ->
219   (Ec__Domains -> Cc__Domains)))
220 semantics__dexp =
221   ((\exp__expressions__344-> case exp__expressions__344 of {
222     (Exp__85 int__notusdefault__144) ->
223       ( (\env__domains__345 ->( (\ec__domains__347 ->
224         (F__114__Domains(\store__domains__349 ->
225           (case ( (case ec__domains__347 of
226             ((F__115__Domains funDomains__352)) -> funDomains__352)
227           ( (D__108__Domains (U__103__Domains
228             (U__110__Domains (D__101__Domains
229             (D__99__NotusDefault int__notusdefault__144)))))))
230               of ((F__114__Domains funDomains__353)) -> funDomains__353)
231                 store__domains__349 ))))));
232     (Exp__86 bool__notusdefault__147) ->
233       ( (\env__domains__354 ->( (\ec__domains__356 ->
234         (F__114__Domains(\store__domains__358 ->
235           (case ( (case ec__domains__356 of
```

```
236             ((F__115__Domains funDomains__361)) -> funDomains__361)
237             ( (D__108__Domains (U__103__Domains
238             (U__109__Domains bool__notusdefault__147))))) of
239             ((F__114__Domains funDomains__362)) -> funDomains__362)
240               store__domains__358 ))))));
241       (Exp__62 ) ->
242         ( (\env__domains__363 ->( (\ec__domains__365 ->
243           (F__114__Domains(\store__domains__367 ->case
244           ( (case store__domains__367 of
245           ((F__113__Domains funDomains__375)) -> funDomains__375)
246           ( (D__100__NotusDefault "input"  ) ) ) of{
247             ((U__126__Domains unused__domains__371)) ->
248               ((E__120__Domains (Enum__0__Error)));
249             ((U__125__Domains(U__106__Domains
250               (L__111__Domains rv__domains__374)))) -> (let {
251               rv__domains__369 = ( util__headRv rv__domains__374 ) ;
252               rv__domains__370 = ( util__tailRv rv__domains__374 ) }
253             in  ((case ((case ec__domains__365 of
254               ((F__115__Domains funDomains__376)) -> funDomains__376)
255               ((D__108__Domains (U__103__Domains ( rv__domains__369 ))))) of
256                 ((F__114__Domains funDomains__377)) -> funDomains__377)
257                 ((F__113__Domains((update (case store__domains__367 of
258                   ((F__113__Domains funDomains__378)) -> funDomains__378)
259                   [((D__100__NotusDefault "input"))]
260                   [((U__125__Domains (U__106__Domains
261                     (L__111__Domains rv__domains__370))))]
262             ))))))}))))));
263       (Exp__61 id__domains__157 ) ->
264         ((\env__domains__379 ->((\ec__domains__381 ->
265           (F__114__Domains(\store__domains__383 ->
266             (case case  ((case env__domains__379 of
267             ((F__112__Domains funDomains__389)) ->
268               funDomains__389) id__domains__157 ) of{
269                 ((U__123__Domains dv__domains__385)) ->
270                   ((case ec__domains__381 of
271                     ((F__115__Domains funDomains__390)) -> funDomains__390)
272                       ((D__108__Domains dv__domains__385)));
273                 _ -> util__err } of
274                   ((F__114__Domains funDomains__391)) ->
275                     funDomains__391) store__domains__383 ))))));
```

```
276     (Exp__63 id__domains__160 exp__expressions__161 ) ->
277       ((\env__domains__392 ->((\ec__domains__394 ->
278         (F__114__Domains(\store__domains__396 ->
279           (case (((semantics__dexp ((Exp__61 id__domains__160)))
280             env__domains__392)
281             (util__isFun ((F__115__Domains
282               ((\ ((D__108__Domains(U__105__Domains fun__domains__398))) ->
283                 ((((semantics__dr exp__expressions__161) env__domains__392)
284                 ((case fun__domains__398 of ((F__118__Domains funDomains__403)
285                 funDomains__403) ec__domains__394))))))))) of
286                   ((F__114__Domains funDomains__404)) ->
287                     funDomains__404) store__domains__396 ))))));
288     (Exp__64 exp__expressions__183 exp__expressions__184
289       exp__expressions__185 ) ->
290       ((\env__domains__405 ->((\ec__domains__407 ->
291         (F__114__Domains(\store__domains__409 ->(case
292           (((semantics__dr exp__expressions__183 ) env__domains__405)
293             (util__isBool ((F__115__Domains
294               ((\ ((D__108__Domains(U__103__Domains(U__109__Domains
295                 bool__notusdefault__411)))) ->
296                   (if ( bool__notusdefault__411 )
297                     then ((((semantics__dexp exp__expressions__184)
298                       env__domains__405 ) ec__domains__407 ) )
299                     else ((((semantics__dexp exp__expressions__185)
300                       env__domains__405 ) ec__domains__407)))))))))
301                       of ((F__114__Domains funDomains__416)) ->
302                       funDomains__416) store__domains__409 ))))));
303     (Exp__60 exp__expressions__200 opr__domains__201 exp__expressions__202 ) -
304       ((\env__domains__417 ->((\ec__domains__419 ->
305         (F__114__Domains(\store__domains__421 ->(case (let {
306           ec__domains__423 = ((F__115__Domains (\ev__domains__424 -> (let {
307           ec__domains__425 = ((F__115__Domains (\ev__domains__426 ->
308             case  ev__domains__424 of{
309               ((D__108__Domains(U__103__Domains rv__domains__428))) ->
310                 case  ev__domains__426 of{
311                   ((D__108__Domains(U__103__Domains rv__domains__429))) ->
312                     ((((semantics__dopr opr__domains__201)rv__domains__428)
313                     rv__domains__429)ec__domains__419)}}))))}
314                     in  (((semantics__dr exp__expressions__202 )
315                       env__domains__417 ) ec__domains__425 )))))}
```

```
316           in  (((semantics__dr exp__expressions__200)env__domains__417)
317             ec__domains__423 ) ) of
318               ((F__114__Domains funDomains__434)) ->
319                 funDomains__434) store__domains__421 ))))))
320 }))
321
322 semantics__dcom :: (Com__Commands -> (Env__Domains ->
323   (Cc__Domains -> Cc__Domains)))
324 semantics__dcom =
325   ((\com__commands__435-> case com__commands__435 of {
326     (Com__66 exp__expressions__211 exp__expressions__212 ) ->
327       ((\env__domains__436 ->( (\cc__domains__438 ->(let {
328         ec__domains__440 = ((F__115__Domains ((\ ev__domains__441 ->
329           case  ev__domains__441 of{
330             ((D__108__Domains(U__102__Domains loc__domains__445))) ->
331               (let {
332                 cc__domains__442 = (((semantics__dr exp__expressions__212)
333                   env__domains__436)
334                 ((util__update loc__domains__445 ) cc__domains__438 ) ) }
335               in  cc__domains__442 )}))))}
336           in  (((semantics__dexp exp__expressions__211) env__domains__436 )
337           (util__isLoc ec__domains__440 )))))));
338       (Com__67 exp__expressions__215 ) ->
339         ((\env__domains__450 ->( (\cc__domains__452 ->
340           (((semantics__dr exp__expressions__215 ) env__domains__450)
341           ((F__115__Domains ( (\ ev__domains__454 ->
342             ((F__114__Domains (\ store__domains__455 ->
343               case  ev__domains__454 of{
344                 ((D__108__Domains(U__103__Domains rv__domains__457))) ->
345                   ((T__119__domains__AnsValue (rv__domains__457 ,
346                     ((case cc__domains__452 of
347                       ((F__114__Domains funDomains__458)) ->
348                     funDomains__458) store__domains__455 ))));
349                  _ -> ( (E__120__Domains (Enum__0__Error)))}))))))))))));
350     (Com__68 id__domains__218 exp__expressions__219 ) ->
351       ((\env__domains__459 ->( (\cc__domains__461 ->
352         (((semantics__dexp ((Exp__61 id__domains__218)))
353           env__domains__459 ) ( util__isProc ( (F__115__Domains
354             ((\ ((D__108__Domains(U__104__Domains proc__domains__463))) ->
355               ((((semantics__dr exp__expressions__219 ) env__domains__459 )
```

```
356               ((case proc__domains__463 of
357                 ((F__117__Domains funDomains__467)) ->
358                   funDomains__467)cc__domains__461 ))))))))))));
359     (Com__69 exp__expressions__222 com__commands__223 com__commands__224 ) ->
360       ((\env__domains__468 ->( (\cc__domains__470 ->
361         ((( semantics__dr exp__expressions__222 ) env__domains__468 )
362         (util__isBool ( (F__115__Domains
363           ((\ ((D__108__Domains(U__103__Domains
364             (U__109__Domains bool__notusdefault__472)))) ->
365           (if (bool__notusdefault__472)
366             then ((((semantics__dcom com__commands__223)
367               env__domains__468 ) cc__domains__470 ))
368             else ((((semantics__dcom com__commands__224 )
369               env__domains__468 ) cc__domains__470 ))))))))))));
370     (Com__70 exp__expressions__227 com__commands__228 ) ->
371       ((\env__domains__477 ->((\cc__domains__479 ->
372         (((semantics__dr exp__expressions__227 ) env__domains__477 )
373         ( util__isBool ((F__115__Domains
374           ((\ ((D__108__Domains(U__103__Domains
375             (U__109__Domains bool__notusdefault__481)))) ->
376               (if (bool__notusdefault__481 )
377                 then ((((semantics__dcom com__commands__228 )
378                   env__domains__477 )
379                   ((((semantics__dcom ( (Com__70 exp__expressions__227
380                     com__commands__228)))
381                     env__domains__477 ) cc__domains__479 ))))
382                 else ( cc__domains__479 ))))))))))));
383     (Com__71 dec__declarations__235 com__commands__236 ) ->
384       ((\env__domains__486 ->((\cc__domains__488 ->(let {
385         dc__domains__490 = ((F__116__Domains(\ env__domains__491 -> (let {
386           env__domains__492 =
387             ((F__112__Domains
388               (\a3 ->
389                 if (case ((case env__domains__491 of
390                   ((F__112__Domains funDomains__498)) -> funDomains__498) a3 )
391                   of{((U__124__Domains unbound__domains__493)) -> True;
392                     _ -> False } )
393                   then (((case env__domains__486 of
394                     ((F__112__Domains funDomains__499)) ->
395                       funDomains__499) a3))
```

```
396                     else (((case env__domains__491 of
397                       ((F__112__Domains funDomains__500)) ->
398                       funDomains__500) a3 )))))}in
399                         (((semantics__dcom com__commands__236 )
400                           env__domains__492 )cc__domains__488)))))}in
401                           (((semantics__ddec dec__declarations__235 )
402                           env__domains__486 ) dc__domains__490 ))))));
403     (Com__65 com__commands__239 com__commands__240 ) ->
404       ((\env__domains__501 ->((\cc__domains__503 ->
405         (((semantics__dcom com__commands__239 ) env__domains__501 )
406         (((semantics__dcom com__commands__240 ) env__domains__501 )
407         cc__domains__503 ))))))
408 }))
409
410 semantics__ddec :: (Dec__Declarations -> (Env__Domains ->
411   (Dc__Domains -> Cc__Domains)))
412 semantics__ddec =
413   ((\dec__declarations__507-> case dec__declarations__507 of {
414     (Dec__73 id__domains__243 exp__expressions__244 ) ->
415       ( (\env__domains__508 ->( (\dc__domains__510 ->
416         (((semantics__dr exp__expressions__244 ) env__domains__508 )
417         ((F__115__Domains ((\ ev__domains__512 -> case  ev__domains__512 of{
418           ((D__108__Domains a5)) ->
419             ((case dc__domains__510 of
420               ((F__116__Domains funDomains__516)) -> funDomains__516)
421                 ((F__112__Domains (update (case util__newEnv of
422                   ((F__112__Domains funDomains__517)) -> funDomains__517)
423                   [ id__domains__243 ]
424                   [((U__123__Domains a5))]))))) ;
425                 _ -> util__err}))))))))));
426     (Dec__74 id__domains__251 exp__expressions__252 ) ->
427       ((\env__domains__518 ->( (\dc__domains__520 ->
428         ((( semantics__dr exp__expressions__252 ) env__domains__518 )
429         (util__ref ((F__115__Domains
430           ((\ ((D__108__Domains(U__102__Domains loc__domains__522))) ->
431             ( ( (case dc__domains__520 of ((F__116__Domains funDomains__526)) ->
432               funDomains__526) ( (F__112__Domains (update (case util__newEnv of
433                 ((F__112__Domains funDomains__527)) -> funDomains__527)
434                 [ id__domains__251 ]
435                 [ ( (U__123__Domains (U__102__Domains loc__domains__522)))]
```

```
436       ))))))))))))))));
437     (Dec__75 id__domains__255 id__domains__256 com__commands__257 ) ->
438       ((\env__domains__528 ->((\dc__domains__530 ->(let {
439         proc__domains__532 = ((F__117__Domains
440           ((\ cc__domains__533 -> ((F__115__Domains
441             (\ ev__domains__534 ->
442               ((case ( util__ref ((F__115__Domains
443                 ((\ ((D__108__Domains(U__102__Domains loc__domains__535))) ->
444                   ((((( semantics__dcom com__commands__257 )
445                   ((F__112__Domains (update (case env__domains__528 of
446                     ((F__112__Domains funDomains__539)) -> funDomains__539)
447                     [ id__domains__256 , id__domains__255 ]
448                     [((U__123__Domains (U__102__Domains loc__domains__535 )))
449                       ((U__123__Domains(U__104__Domains proc__domains__532)))]
450                   )))) cc__domains__533 )))))))
451                     of ((F__115__Domains funDomains__540)) ->
452                       funDomains__540) ev__domains__534)))))))}in
453                       ((case dc__domains__530 of
454                         ((F__116__Domains funDomains__541)) -> funDomains__541
455                         ((F__112__Domains (update (case util__newEnv of
456                         ((F__112__Domains funDomains__542)) -> funDomains__542
457                         [id__domains__255]
458                         [( (U__123__Domains(U__104__Domains proc__domains__532
459       ))))))))));
460     (Dec__76 id__domains__260 id__domains__261 exp__expressions__262 ) ->
461       ((\env__domains__543 ->((\dc__domains__545 ->(let {
462         fun__domains__547 = ((F__118__Domains ((\ ec__domains__548 ->
463           ((F__115__Domains (\ ev__domains__549 ->
464             ((case ( util__ref ( (F__115__Domains
465               ((\ ((D__108__Domains(U__102__Domains loc__domains__550))) ->
466                 ((((semantics__dexp exp__expressions__262)
467                 ((F__112__Domains (update (case env__domains__543 of
468                   ((F__112__Domains funDomains__554)) ->
469                   funDomains__554)
470                   [ id__domains__261 , id__domains__260 ]
471                   [((U__123__Domains (U__102__Domains loc__domains__550))) ,
472                    ((U__123__Domains (U__105__Domains fun__domains__547)))])))
473             ec__domains__548)))))) of ((F__115__Domains funDomains__555)) ->
474                 funDomains__555) ev__domains__549 )))))))}in
475                  ((case dc__domains__545 of
```

```
476                   ((F__116__Domains funDomains__556)) ->
477                     funDomains__556) ((F__112__Domains
478                       (update (case util__newEnv of
479                       ((F__112__Domains funDomains__557)) -> funDomains__557)
480                       [id__domains__260]
481                       [((U__123__Domains (U__105__Domains fun__domains__547)))])
482         ))))));
483     (Dec__72 dec__declarations__265 dec__declarations__266 ) ->
484       ((\env__domains__558 ->((\dc__domains__560 ->
485         (((semantics__ddec dec__declarations__265 ) env__domains__558)
486         ((F__116__Domains ( (\ env__domains__562 -> (let {
487           env__domains__565 = ((F__112__Domains ((\ a5 ->
488             if ( case ((case env__domains__562 of
489               ((F__112__Domains funDomains__572)) ->
490                 funDomains__572) a5 ) of{
491                 ((U__124__Domains unbound__domains__567)) -> True;
492               _ -> False } )
493               then (((case env__domains__558 of
494                 ((F__112__Domains funDomains__573)) ->
495                   funDomains__573) a5))
496               else (((case env__domains__562 of
497                 ((F__112__Domains funDomains__574)) ->
498                   funDomains__574) a5))))))}
499                     in  (((semantics__ddec dec__declarations__266)
500                       env__domains__565) ((F__116__Domains
501                         ((\ env__domains__563 ->
502                         (let {
503                           env__domains__564 = ((F__112__Domains ((\ a3 ->
504                             if ( case  ( (case env__domains__563 of
505                               ((F__112__Domains funDomains__575)) ->
506                                 funDomains__575) a3 ) of{
507                                 ((U__124__Domains unbound__domains__570)) -> True;
508                                 _ -> False } )
509                               then (((case env__domains__562 of
510                                 ((F__112__Domains funDomains__576)) ->
511                                   funDomains__576) a3))
512                               else (((case env__domains__563 of
513                                 ((F__112__Domains funDomains__577)) ->
514                                   funDomains__577) a3))))))}in
515                                   ((case dc__domains__560 of
```

```
516                                        ((F__116__Domains funDomains__578)) ->
517                                          funDomains__578) env__domains__564)))))))
518 }))
519
520 semantics__initialStore :: (Loc__Domains -> StoreValue__Domains)
521 semantics__initialStore =
522   ((\loc__domains__579-> ((U__126__Domains(E__122__Domains(Enum__2__Unused))))
523
524 util__cont :: (Ec__Domains -> Ec__Domains)
525 util__cont =
526   ((\ec__domains__580->
527     (F__115__Domains(\ev__domains__581 ->
528       (F__114__Domains(\store__domains__583 ->case  ev__domains__581 of{
529         ((D__108__Domains(U__102__Domains loc__domains__585))) ->
530           case  ( ( (case store__domains__583 of
531             ((F__113__Domains funDomains__588)) ->
532               funDomains__588) loc__domains__585 ) ) of{
533               ((U__125__Domains(U__107__Domains rv__domains__586))) ->
534                 ((case ((case ec__domains__580 of
535                   ((F__115__Domains funDomains__589)) -> funDomains__589)
536                   ((D__108__Domains (U__103__Domains rv__domains__586))))
537                   of ((F__114__Domains funDomains__590)) ->
538                   funDomains__590) store__domains__583 ) ;
539               ((U__126__Domains unused__domains__587)) ->
540                 ((E__120__Domains (Enum__0__Error  ) ) ) ;
541               _ -> ( (E__120__Domains (Enum__0__Error  ) ) ) } ;
542         _ -> ( (E__120__Domains (Enum__0__Error ))) }))))))
543
544 util__update :: (Loc__Domains -> (Cc__Domains -> Ec__Domains))
545 util__update =
546   ((\loc__domains__591-> ( (\cc__domains__274 ->
547     (F__115__Domains(\ev__domains__275 ->(F__114__Domains
548       (\store__domains__596 ->case  ev__domains__275 of{
549         ((D__108__Domains(U__103__Domains a2))) ->
550           ((case cc__domains__274 of ((F__114__Domains funDomains__598)) ->
551             funDomains__598) ((F__113__Domains
552               (update (case store__domains__596 of
553                 ((F__113__Domains funDomains__599)) -> funDomains__599)
554                 [ loc__domains__591 ]
555                 [((U__125__Domains (U__107__Domains a2)))]))))) ;
```

```
556         _ -> ( (E__120__Domains (Enum__0__Error  ) ) ) } ))))))))
557
558 util__deref :: (Ec__Domains -> Ec__Domains)
559 util__deref =
560   ( (\ec__domains__600-> (F__115__Domains
561     (\ev__domains__601 ->(F__114__Domains
562       (\store__domains__603 ->case  ev__domains__601 of{
563         ((D__108__Domains(U__102__Domains loc__domains__606))) ->
564           ( (case ( (case ( util__cont ec__domains__600 ) of
565             ((F__115__Domains funDomains__607)) -> funDomains__607)
566               ( (D__108__Domains (U__102__Domains loc__domains__606 )))) of
567                 ((F__114__Domains funDomains__608)) ->
568                   funDomains__608) store__domains__603 ) ;
569         _ -> ( (case ( (case ec__domains__600 of
570           ((F__115__Domains funDomains__609)) -> funDomains__609)
571             ev__domains__601 ) of
572             ((F__114__Domains funDomains__610)) -> funDomains__610)
573               store__domains__603 )}))))))
574
575 util__new :: (Store__Domains -> Loc__Domains)
576 util__new = ( (\store__domains__611->
577   ((util__findNew store__domains__611 ) ((D__100__NotusDefault "" )))))
578
579 util__findNew :: (Store__Domains -> (Loc__Domains -> Loc__Domains))
580 util__findNew =
581   ((\store__domains__613->
582     ((\((D__100__NotusDefault string__notusdefault__282)) ->
583       if ( case  ( ( (case store__domains__613 of
584         ((F__113__Domains funDomains__619)) -> funDomains__619)
585           ((D__100__NotusDefault string__notusdefault__282  ) ) ) ) of{
586             ((U__126__Domains unused__domains__616)) -> True;
587             _ -> False } )
588             then (((D__100__NotusDefault string__notusdefault__282)))
589             else ( (let {
590               a0 = ( string__notusdefault__282  ++  "x" ) }
591                 in  ( ( util__findNew store__domains__613 )
592                   ( (D__100__NotusDefault a0)))))))))
593
594 util__ref :: (Ec__Domains -> Ec__Domains)
595 util__ref =
```

```
596   ((\ec__domains__620->
597     (F__115__Domains(\ev__domains__621 ->
598       (F__114__Domains(\store__domains__623 ->case
599         (( util__new store__domains__623 ) ) of{
600           ( loc__domains__627) -> ( (case ( (case
601             ((util__update loc__domains__627 )
602               (((case ec__domains__620 of((F__115__Domains funDomains__628)) -
603                 funDomains__628) ((D__108__Domains
604                   (U__102__Domains loc__domains__627))))))
605               of ((F__115__Domains funDomains__629)) -> funDomains__629)
606                 ev__domains__621 ) of ((F__114__Domains funDomains__630)) ->
607                   funDomains__630) store__domains__623 )}))))))
608
609 util__err :: Cc__Domains
610 util__err =
611   (F__114__Domains(\store__domains__631->
612     ((E__120__Domains(Enum__0__Error )))))
613
614 util__newEnv :: Env__Domains
615 util__newEnv =
616   ((F__112__Domains ((\ a0 ->
617     ( (U__124__Domains (E__121__Domains (Enum__1__Unbound  ) ) ) ) ) ) )))
618
619 util__headRv :: ([Rv__Domains] -> Rv__Domains)
620 util__headRv =
621   ( \( ( (rv__domains__287:rv__domains__634)) ) ->rv__domains__287 )
622
623 util__tailRv :: ([Rv__Domains] -> [Rv__Domains])
624 util__tailRv =
625   ( \( ( (rv__domains__288:rv__domains__636)) ) ->rv__domains__636 )
626
627 util__isRv :: (Ec__Domains -> Ec__Domains)
628 util__isRv =
629   ( (\ec__domains__637->
630     (F__115__Domains(\ev__domains__638 ->
631       if ( case  ev__domains__638 of{
632         ((D__108__Domains(U__103__Domains rv__domains__640))) -> True;
633         _ -> False } )
634         then
635           (((case ec__domains__637 of
```

```
636             ((F__115__Domains funDomains__642)) -> funDomains__642)
637               ev__domains__638))
638         else ( util__err ) ))))
639
640 util__isFun :: (Ec__Domains -> Ec__Domains)
641 util__isFun =
642   ( (\ec__domains__643-> (F__115__Domains
643     (\ev__domains__644 ->
644       if ( case  ev__domains__644 of{
645         ((D__108__Domains(U__105__Domains fun__domains__646))) -> True;
646         _ -> False } )
647         then (((case ec__domains__643 of
648           ((F__115__Domains funDomains__648)) -> funDomains__648)
649           ev__domains__644 ) ) else ( util__err ) ))))
650
651 util__isBool :: (Ec__Domains -> Ec__Domains)
652 util__isBool =
653   ( (\ec__domains__649-> (F__115__Domains(\ev__domains__650 ->
654     if ( case  ev__domains__650 of{
655         ((D__108__Domains(U__103__Domains(U__109__Domains
656           bool__notusdefault__652)))) -> True;
657         _ -> False } )
658         then ( ( (case ec__domains__649 of
659           ((F__115__Domains funDomains__654)) -> funDomains__654)
660             ev__domains__650 ) )
661         else ( util__err ) ))))
662
663 util__isLoc :: (Ec__Domains -> Ec__Domains)
664 util__isLoc =
665   ( (\ec__domains__655-> (F__115__Domains
666     (\ev__domains__656 ->
667       if ( case  ev__domains__656 of{
668         ((D__108__Domains(U__102__Domains loc__domains__658))) -> True;
669         _ -> False } )
670       then ( ( (case ec__domains__655 of
671         ((F__115__Domains funDomains__660)) -> funDomains__660)
672           ev__domains__656 ) )
673       else ( util__err ) ))))
674
675 util__isProc :: (Ec__Domains -> Ec__Domains)
```

```
676 util__isProc =
677   ( (\ec__domains__661->
678     (F__115__Domains(\ev__domains__662 ->if ( case  ev__domains__662 of{
679       ((D__108__Domains(U__104__Domains proc__domains__664))) -> True;
680       _ -> False } )
681       then ( ( (case ec__domains__661 of
682         ((F__115__Domains funDomains__666)) -> funDomains__666)
683           ev__domains__662 ) )
684       else ( util__err ) ))))
685
686
687
688
689 main = do
690   (x:xs) <- getArgs
691   progL <- readFile x
692   inL <- sReaderNotus (head(matchInFiles [(Stdin)] xs))
693   sWriterNotus (head(matchOutFiles [(Stdout)] xs))
694     (main__dmain  (parse (alexScanTokens progL)) inL)
```

Listagem D.13: Módulo principal do interpretador *Small*

Referências Bibliográficas

[Berk, 2003] Berk, E. (2003). JLex: A lexical analyzer generator for Java. Home page: `http://www.cs.princeton.edu/~appel/modern/java/JLex/`.

[Brown et al., 1992] Brown, D.; Levine, J. e Mason, T. (1992). *lex & yacc (A Nutshell Handbook)*. O'Reilly Media.

[Cardelli e Wegner, 1986] Cardelli, L. e Wegner, P. (1986). On Understanding Types,Data Abstraction, and Polymorphism. *ACM Computing Surveys 17*, 5:471–522.

[Cormen, 2003] Cormen, T. H. (2003). Using the clrscode Package in LATEX2. Home page: `http://www.cs.dartmouth.edu/~thc/clrscode/clrscode.pdf`.

[Gordon, 1979] Gordon, M. J. (1979). *The Denotational Description of Programming Languages - An Introduction*. Springer-Verlag.

[Gurevich, 1995] Gurevich, Y. (1995). Evolving algebras 1993: Lipari guide. *Specification and validation methods*.

[Hoare, 1969] Hoare, C. A. R. (1969). An Axiomatic Basis for Computer Programming. *Communications of the ACM, 12.10*.

[Jones, 2003] Jones, S. P. (2003). Haskell 98 language and libraries the revised report. Technical report, Cambridge University Press.

[Kiczales et al., 1997] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.-M. e Irwin, J. (1997). Aspect-oriented programming. *Proceedings of the 11th European Conference on Object-Oriented Programming.*

[Laddad, 2003] Laddad, R. (2003). *AspectJ in Action*. Manning Publications.

[Liang, 1998] Liang, S. (1998). *Modular Monadic Semantics and Compilation*. Ph.d. thesis, Yale University. adviser-Paul Hudak.

[Liang e Hudak, 1996] Liang, S. e Hudak, P. (1996). Modular Denotational Semantics for Compiler Construction. *6th European Symposium on Programming Languages and Systems*.

[Liang et al., 1995] Liang, S.; Hudak, P. e Jones, M. (1995). Monad Transformers and Modular Interpreters. *Conference Record of POPL'95: 22nd ACM SIGPLAN-SICACT Symposium on Principles of Programming Languages*.

[Marlow, 2005a] Marlow, S. (2005a). Alex: A lexical analyser generator for Haskell. Home page: `http://www.haskell.org/alex/`.

[Marlow, 2005b] Marlow, S. (2005b). Happy: The Parser Generator for Haskell. Home page: `http://www.haskell.org/happy/`.

[Marlow, 2005c] Marlow, S. (2005c). The Glasgow Haskell Compiler. Home page: `http://www.haskell.org/ghc/`.

[Meyer, 1997] Meyer, B. (1997). *AspectJ in Action*. Prentice Hall.

[Mosses, 1988] Mosses, P. D. (1988). The Modularity of Action Semantics. *Technical Report DAIMI IR-75*.

[Mosses, 1996a] Mosses, P. D. (1996a). A Tutorial On Action Semantics. *Tutorial notes for FME'96: Formal Methods Europe*.

[Mosses, 1996b] Mosses, P. D. (1996b). Theory and Practice of Action Semantics. *Proceedings of the 21st International Symposium on Mathematical Foundations of Computer Science*.

[Mosses, 2001] Mosses, P. D. (2001). What Use Is Formal Semantics? *PSI Report, Perspective of System Informatics 2001*.

[Mosses, 2005] Mosses, P. D. (2005). Modular Structural Operational Semantics. *J. Logic and Algebraic Programming*.

[Mosses e Watt, 1986] Mosses, P. D. e Watt, D. A. (1986). The Use of Action Semantics. *Technical Report DAIMI PB-217*.

[Mosses e Watt, 1993] Mosses, P. D. e Watt, D. A. (1993). Pascal Action Semantics, Version 0.6. Disponível em ftp://ftp.brics.dk/Projects/AS/Papers/MossesWatt93DRAFT/.

[Moura et al., 2002] Moura, H.; Menezes, L. C.; Monteiro, M.; Sampaio, P. e Cansanção, W. (2002). The Abaco System: an Action Tool for Programming Language Designers.

[Moura, 1996] Moura, H. P. (1996). An Overview of Action Semantics. *I Simpósio Brasileiro de Linguagens de Programação*.

[Plotkin, 1981] Plotkin, G. (1981). A Structural Approach to Operational Semantics. *Technical Report, DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark*.

[Scott, 1976] Scott, D. (1976). Data type as Lattices. *SIAM J. on Comp.*, 5:522–587.

[Sedgewick, 1990] Sedgewick, R. (1990). *Algortithms in C*. Addison-Wesley Publishing Company.

[Sipser, 1996] Sipser, M. (1996). *Introduction to the Theory of Computation*. International Thomson Publishing.

[Slonneger, 1995] Slonneger, K. (1995). *Formal Syntax and Semantics of Programming Languages - A Laboratory Based Approach*. Addison-Wesley.

[Stoy, 1977] Stoy, J. (1977). Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. *MIT Press, Cambridge, MA*.

[Sudkamp, 1997] Sudkamp, T. (1997). *Languages and Machines*. Addison-Wesley. 2a edição.

[The GHC Team, 2006] The GHC Team (2006). The Glorious Glasgow Haskell Compilation System User's Guide. Home page: `http://www.haskell.org/ghc/docs/latest/html/users_guide/index.html`. Versão 6.6.

[Thompson, 1999] Thompson, S. (1999). *Haskell: The Craft of Functional Programming*. Addison-Wesley, 2a edição edio.

[Tirelo, 2005] Tirelo, F. (2005). *Semântica Multidimensional de Linguagens de Programação*. Proposta de tese de doutorado, Universidade Federal de Minas Gerais. orientador: Roberto da Silva Bigonha.

[Tirelo e Bigonha, 2006] Tirelo, F. e Bigonha, R. S. (2006). Notus. Technical Report LLP 001/2007, Laboratório de Linguagens de Programação, UFMG.

[Triola, 1998] Triola, M. (1998). *Introdução à Estatística*. LTC. Tradução da 7ª edição.

[van Deursen e Mosses, 1996] van Deursen, A. e Mosses, P. D. (1996). ASD: The Action Semantic Description Tools. Home page: `http://www.brics.dk/Projects/AS/Tools/ASD/`.

[Wadler, 1992] Wadler, P. (1992). The essence of functional programming. *19'th Annual Sympoium on Principles of Programming Languages*.

[Zhang e Xu, 2004] Zhang, Y. e Xu, B. (2004). A survey of semantic description frameworks for programming languages. *ACM SIGPLAN Notices*.