Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Bacharelado em Ciência da Computação

Manual do Usuário de Notus[1]

Mirlaine Aparecida Crepalde
Roberto da Silva Bigonha
Fábio Tirelo
Tays Cristina do Amaral P. S.

LLP02/2008

[1]**Projeto desenvolvido com apoio da FAPEMIG Processo CEX-1484/06**

Sumário

1 Introdução

Este manual tem o objetivo de descrever a linguagem puramente funcional *Notus*, apresentando-a através de exemplos. *Notus*, definida em [6], é uma linguagem de domínio específico que implementa a metodologia de Semântica Denotacional Multidimensional para descrição formal da semântica de linguagem de programação. A Semântica Multidimensional é uma abordagem nova de formulação de Semântica Denotacional que permite o estruturamento modular e incremental de definição formal de linguagens de programação.

Em *Notus* é possível definir características léxicas, sintáticas e semânticas de linguagens de programação de forma modular e incremental. A linguagem provê cláusulas específicas para extensão da gramática concreta e domínios da linguagem, além das **transformações** [5] que permitem alterar declarações das funções semânticas, possibilitando uma escrita modular e incremental sem alteração nos módulos já existentes.

O primeiro compilador de *Notus*, descrito em [1], foi desenvolvido em Java e traduz todas as construções de *Notus* para a linguagem *Haskell*, de propósito geral. Ele é disponibilizado tanto para a plataforma Windows quanto para a Linux. A seção 8 apresenta como obter e instalar esse compilador para escrever definições utilizando a linguagem *Notus*.

As informações do manual são baseadas em uma coletânea de informações sobre a linguagem *Notus* contidas em [1], [6] e [5]. Além disso, novas informações e exemplos são fornecidos ao usuário.

2 Convenções Léxicas de Notus

2.1 Definição de Nomes

Os nomes para pacotes, módulos e domínios podem conter letras, números, além do caracter "_", entretanto devem sempre iniciar com uma letra maiúscula e finalizarem com uma letra, maiúscula ou minúscula. Exemplos desses nomes são:

```
DominioA Dominiob Dominio_c Dominio_D DOMINIO_e Pacote1M P1m P1_M
```

Os nomes de *tokens* e *elements*, na especificação léxica, variáveis, na especificação sintática, funções, padrões de identificador, enumerações e constantes podem conter letras, números, além do caractere "_", entretanto, devem iniciar com uma letra minúscula ou "_". Sequências de dígitos no final desses nomes são consideradas decorações. Assim, **comando123** é considerado ser o nome **comando** decorado com **123**. O domínio de um desses elementos pode ser deduzido implicitamente por eliminar a decoração e capitalizar sua primeira letra. Se existir um domínio com tal nome, esse componente é considerado do domínio. Assim, **comando1234** é considerado deste domínio **Comando**, caso ele exista. Da mesma forma, **int1** e **int2** são considerados do domínio **Int**.

Exemplos desses nomes são:

```
f f1 f_ functionUm elem_a elem_A1
```

É considerado erro definir nomes com dois ou mais símbolos *underline*, "_", adjacentes.

2.2 Constantes Literais

Notus suporta as seguintes constantes literais:

- **true** e **false** para denotar valores booleanos;
- inteiros de 8, 16 e 32 bits escritos nas seguintes bases:
  - decimal: uma sequência de um ou mais dígitos de 0 a 9. Exemplos: 0, 10, 98501;

  - octal: sequência de um ou mais dígitos de 0 a 7, precedidos pelo dígito 0. Exemplos: 012, 07;
  - hexadecimal: sequência e um ou mais dígitos de 0 a 9 e letras de **a** a **f**, precedidas por **0x** ou **0X**. Exemplos: 0x45f , 0Xffff, 0x1111

- inteiros de 64 bits para denotar inteiros longos, escritos nas seguintes bases:
  - decimal: uma sequência de um ou mais dígitos de 0 a 9 finalizados com a letra **l** ou **L**. Exemplos: 0l, 10L, 98501L;
  - octal: sequência de um ou mais dígitos de 0 a 7, precedidos pelo dígito 0 e finalizados com a letra **l** ou **L**. Exemplos: 012l, 07L;
  - hexadecimal: sequência e um ou mais dígitos de 0 a 9 e letras de **a** a **f**, precedida por **0x** ou **0X** e finalizada com a letra **l** ou **L**. Exemplos: 0x45fl , 0XffffL, 0x1111l

- números de ponto flutuante de 64 bits, descritos pelo **IEEE 754**, para denotar valores de precisão dupla. Exemplos: 1, 0.1, 0.1E5, 0.22e-10, 1.25e+5;
- números de ponto flutuante de 32 bits, descritos pelo **IEEE 754**, para denotar valores de ponto flutante. Exemplos: 1f, 0.1F, 0.1E5f, 0.22e-10F, 1.25e+5F;
- caracteres entres aspas simples para denotar caracteres. Exemplos: 'a', 'b', '1';
- sequência de caracteres entres aspas duplas para denotar *Strings*. Exemplo: "notus"
- sequências de escape:

```
\a alarme
\b backspace
\f form-feed
\n line-feed
\t
\\ backslash
\' aspas simples
\" aspas duplas
```

2.3 Comentários

Linhas de comentários em *Notus* iniciam com "//". Tudo que segue o "//"até o fim da linha é ignorado pelo compilador. Os comentários também podem vir entre "/*"e "*/". O compilador também ignora tudo entre esses símbolos.

Exemplos de comentários em *Notus*:

```
//Isso e um comentario em Notus

/*Os comentarios tambem podem ser feitos dessa
  forma*/
```

3 Tipos em Notus

Os tipos de dados possíveis em *Notus* são denominados domínios. Isso porque *Notus* é uma linguagem funcional específica para definição de linguagens, sendo essa nomeclatura, portanto, conveniente. Assim, quando a palavra domínio for nesse manual utilizada, deve-se entender que está se referindo a tipos. Os domínios de *Notus* são domínios definidos no trabalho de Scott, descrito em [4]. Domínios de *Scott* conseguem modelar semântica de não-terminação, permitindo, por exemplo, encontrar o menor ponto fixo de equações recursivas da forma $f(x) = f(x + 1)$. O usuário não precisa conhecer essa teoria de domínios para utilizar a linguagem *Notus* e prosseguir na leitura deste manual. Para maiores detalhes sobre essa teoria ver a referência citada.

3.1 Domínios Sintáticos

Domínios sintáticos denotam tipos de *tokens* e variáveis da gramática concreta e podem ser explicitamente declarados ou automaticamente coletados. Em *Notus* domínios sintáticos devem iniciar com letra maiúscula. Se um *token* ou variável da gramática não especifica um domínio, o nome do domínio sintático desse elemento é obtido ao capitalizar a primeira letra do seu identificador. Adicionalmente, identificadores de *tokens* e variáveis podem ser "decorados"por uma sequência de dígitos.

```
token fun = ...
token fun1 = ...
token var = ...
token num : Exp = ...
exp ::= ...
primary : Exp ::= ...
```

Nesse exemplo, os domínios **Fun**, **Var** e **Exp** são automaticamente detectados da definição dos *tokens* **fun** e **var** e da variável **exp**. Além disso, o domínio sintático para **num** e **primary** é definido para ser **Exp**. O domínio do *token* **fun1**, **Fun**, é obtido capitalizando a primeira letra do identificador do *token* e removendo a sua "decoração".

Domínios sintáticos podem ser definidos com controle de visibilidade privado, público ou de pacote. Um domínio sintático é definido como público quando a visibilidade é especificada com a palavra **public**, podendo ser acessado de qualquer módulo da definição.

Um domínio sintático é definido como privado quando a visibilidade é especificada com a palavra **private**, podendo ser acessado somente dentro do módulo onde foi definido. Por fim, um domínio sintático é definido como de pacote quando nenhuma palavra específica é utilizada para visibilidade, podendo ser acessado dentro de todos os módulos que pertencem ao mesmo pacote do módulo em que são definidos. Para saber melhor sobre módulos e pacotes de *Notus* ver o capítulo 4. Domínios públicos e privados são definidos com o uso da cláusula **syntactic**, da seguinte forma:

visibilidade **syntactic** *IdentificadorDoDomínio*;

O exemplo a seguir define os domínios **Id** e **Num** como privados e o domínio **Exp** como público.

```
public syntactic Exp;
private syntactic Id, Num;
```

Notus possui os seguintes domínios sintáticos pré-definidos:

- *Bool*: representa o conjunto **false** e **true**;
- *Int*: representa o domínio dos inteiros de 32 bits;
- *Long*: representa o domínio dos 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.

3.2 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 da seguinte forma:

visibilidade **IdentificadorDoDominio** = *Expressão* ";"

O identificador do domínio semântico, assim como do domínio sintático, deve iniciar com letra maiúscula. A visibilidade pode ser *public* para o domínio ser visível em todos os módulos da especificação, *private* para o domínio ser visível somente no módulo em que é declarado ou de pacote quando não há declaração explícita de visibilidade.

O exemplo abaixo mostra a definição dos domínios semânticos **Value**, **Loc** e **State**:

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

Os 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 expressões para definir os domínios semânticos são compostas por operações realizadas sobre os domínios básicos, que são os domínios sintáticos, os domínios pré-definidos ou enumeração de domínios.

Os domínios sintáticos e pré-definidos já foram descritos anteriormente. A enumeração de domínios permite definir conjuntos de constantes, e cada elemento de uma enumeração é um elemento iniciando com letra minúscula. O seguinte exemplo mostra a definição dos domínios **Color** e **Language** utilizando enumeração:

```
public Cor = {vermelha, verde, azul} ;
private Tamanho = {pequeno, medio, grande} ;
Linguagem = {java, haskell, notus} ;
```

As operações usadas nas expressões para definição de tipos são:

1. produto cartesiano: para construção de tuplas;
2. união disjunta: para construção de soma separada;
3. recursão: para construção de domínios recursivos, por exemplo, listas;
4. construtor de domínios funcionais.

3.2.1 Tipo Tupla

O domínio(tipo) tupla é obtido com a operação de produto cartesiano dos domínios que o compõem. As tuplas devem ser identificadas com um construtor de tipo, e seus domínios podem ser recursivamente definidos. Se um domínio é definido por apenas uma expressão tupla, seu construtor pode ser omitido. Segue um exemplo de criação de domínios de tuplas:

```
Par = (Int, Bool) ;
ArvoreSimples = Nada () | Um (Int) | Dois (Int,Int) | Tres (Int, Int, Int) ;
ArvoreRecursiva = Folha Int | Ramo (Int, ArvoreRecursiva, ArvoreRecursiva) ;
```

O domínio **Par** representa $IntxBool$ e o seu construtor é implicitamente obtido como **Par**. O domínio **ArvoreSimples** representa $() + Int + (IntxInt) + (IntxIntxInt)$. O domínio **ArvoreRecursiva** define recursivamente uma árvore binária não-vazia.

3.2.2 Tipo Lista

O domínio lista representa uma lista de elementos de um domínio. O seguinte exemplo mostra a definição do domínio **Input** como uma lista de valores pertencentes ao domínio **String**:

```
Input = String* ;
```

3.2.3 União Disjunta

A operação de união disjunta permite definir novos tipos como a soma separada de domínios e possui a seguinte forma:

$$d_1|d_2|...|d_n,$$

onde cada d_i é um domínio. O exemplo seguinte mostra a definição do domínio **Value** como soma separada dos domínios **Int**, **Bool** e **{error}**:

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

3.2.4 Funções

Por fim a quarta operação citada anteriormente permite a construção de domínios funcionais que nada mais são que uma função de um domínio para outro e possuem a seguinte forma:

$$d_1 \rightarrow d_2,$$

em que d_1 e d_2 são domínios. O seguinte exemplo mostra a criação dos domínios funcionais **State** e **Econt**:

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

O operador $\rightarrow$ é associativo à direita e, por isso, nesse exemplo o domínio **Econt** representa o domínio $Value \rightarrow (State \rightarrow State)$.

3.2.5 Precedência de Operações para Definir Tipos

A tabela seguinte mostra a precedência das operações para definir tipos em *Notus*:

Tabela 1: *Precedência das Operações*

Maior precedência	formação de tupla
	recursão, para construção de lista
	construção de domínios funcionais
Menor precedência	união disjunta

4 Organização da Definição

4.1 Módulos

Notus prevê a decomposição da definição de uma linguagem em módulos, favorecendo a extensibilidade e legibilidade. Um módulo em *Notus* encapsula a descrição de algumas características de uma linguagem e é composto pelos seguintes elementos:

- importações, em que enumeram-se as dependências do módulo em relação a outros módulos;
- especificações léxicas e sintáticas, que definem ou estendem os componentes léxicos e sintáticos de um conjunto de construções;
- definição ou extensão de domínios sintáticos e semânticos;
- declaração e definição de funções.

Segue um exemplo da definição do módulo **ModuloA** com seus elementos:

```
module ModuloA

  import ModuloB ;  //Secao de importacao

  prog ::= "teste" ;  //Definicao de elemento sintatico

  Entrada = Int ;  //Definicao de dominio semantico

  Saida = String ;  //Definicao de dominio semantico

  function funcaoTeste : Entrada -> Saida ;  //Declaracao de funcao
  funcaoTeste entrada = entradaToSaida entrada ;  //definicao de funcao

end
```

No exemplo acima, **entradaToSaida** é uma função declarada no módulo **ModuloB**. Cada módulo deve ser definido em um arquivo separado, cujo nome deve ser o nome principal do módulo, sem qualificação (o caminho completo do arquivo dentro da hierarquia de pacotes, a ser descrita na seção 4.2), e com extensão **nts**. Logo o módulo **ModuloA** deve ser definido no arquivo *ModuloA.nts*.

Em cada seção do módulo é possível definir a visibilidade dos identificadores como públicos, privados ou de pacote. Um componente público, definido como **public**, é visível em todos os módulos da definição. Um componente privado, definido como **private**, é visível somente no módulo onde é definido. Por fim, se nenhuma visibilidade é especificada, a visibilidade do componente é de pacote e ele é somente visível pelos módulos dentro do pacote que contém o módulo onde o componente é definido.

Supondo que o módulo **ModuloC** seja definido no pacote **PacoteA**, dentro do arquivo *ModuloC.nts*:

```
module PacoteA.ModuloC

 public Tipo = Int | Bool;
 Id = String;

 private function auxiliar : Id -> Tipo;

...

end
```

O domínio **Tipo** é definido como público, portanto, é acessível em todos os módulos da definição. O domínio **Id** é definido com visibilidade de pacote, portanto, é acessível em todos os módulos dentro do pacote **PacoteA**. Por fim, a função **auxiliar** é definida como privada, sendo, assim, acessível somente no módulo **ModuloC**.

Se um módulo M_1 importa um módulo M_2, então todos os componentes de M_2 definidos como público podem ser usados em M_1 sem qualificadores. Componentes privados de M_2 não são acessíveis em M_1. É possível também usar o nome completo do componente público do módulo, sem necessidade de importá-lo. Considerando o módulo **ModuloC** definido anteriormente:

```
module ModuloD
 import PacoteA.ModuloC;

 TipoGeral = Tipo | String;
```

```
...

end

module ModuloD
 TipoGeral = PacoteA.ModuloC.Tipo | String;
 ...
end
```

Os dois exemplos acima exemplificam o uso de componentes públicos do módulo **ModuloC** pelo módulo **ModuloD**. No primeiro exemplo, os componentes públicos são utilizados sem qualificação, dado que o módulo **ModuloC** foi importado. No segundo exemplo, como o módulo **ModuloC** não foi importado, o domínio **Tipo** é usado com nome qualificado.

Quando há conflitos de nomes nos módulos sendo importados, é preciso que o módulo que faz a importação qualifique cada uso do nome em conflito. O exemplo seguinte ilustra as possíveis situações:

```
module A
public function f : Int -> Int;
...
end

module B
public function f : Int -> Int;
...
end

module C
import A;
  function g : Int -> Int;
  g x = f x + B.f x;  //Correto
...
end

module C
import A,B
  function g : Int -> Int;
```

```
  g x = A.f x + B.f x;  //Correto
...
end

module C
  function g : Int -> Int;
  g x = A.f x + B.f x;  //Correto
...
end

module C
import A,B
  function g : Int -> Int;
  g x = f x + f x;  //Incorreto. Deveria ser A.f x + B.f x
...
end
```

4.1.1 Módulo Principal

O módulo principal de uma especificação em *Notus* é denominado **Main**. Ele é obrigatório e precisa ser definido no arquivo *Main.nts*. Nele, são especificados os seguintes elementos:

1. função de pré-processamento do arquivo de entrada;
2. símbolo inicial da gramática;
3. arquivos de entrada;
4. arquivos de saída;
5. função semântica de avaliação da Árvore Sintática Abstrata, *AST*.

A função de pré-processamento é opcional. Ela é do tipo $String \rightarrow String$ e transforma a *String* que representa as linhas de um programa, a ser interpretado pelo compilador gerado para a linguagem definida em *Notus*, em uma nova *String*. Essa função permite a especificação em *Notus* de linguagens que definem regras de simplificação de sintaxe, como *Haskell*, que possibilita ao programador a omissão de alguns delimitadores, que podem ser deduzidos pelo contexto. A função de pré-processamento é definida pela cláusula **prepoc function-name**, em que **function-name** é o nome de uma função definida no módulo *main* ou em outro módulo importado por ele. Um exemplo do uso da cláusula *preproc* segue:

```
module Main
...
 preproc Preprocessing.preprocessing;

...
end
```

Considerando a especificação de uma linguagem em que cada linha é um comando de atribuição e que não há necessidade de escrever ponto-e-vírgula no final da linha. Segue o módulo **Preprocessing** que contém a função **preprocessing** para transformar um programa escrito nessa linguagem em outro que contém um ponto-e-virgula ao fim de cada linha.

```
module Preprocessing
  public function preprocessing: String -> String ;
  preprocessing s = lhs s;

  function hls: String -> String ;
  lhs () = ();
  lhs '\n':s1 = ';':('\n':(lhs s1)) ;
  lhs c:s1 = c:(lhs s1) ;
end
```

O símbolo inicial da gramática é definido por meio da cláusula **syntax s**, onde s é uma variável da gramática concreta que pertence ao domínio S, domínio da raiz da **AST**, *Abstract Syntax Tree*, produzida pelo analisador sintático. Caso o símbolo inicial da gramática não seja definido, uma advertência de compilação é gerada.

Os valores de entrada correspondem a *Strings* que podem ser lidas da entrada padrão ou de arquivos textos. Eles são definidos por meio da cláusula, opcional, *input* $in_1, in_2, ..., in_n$, em que cada in_i pode ser:

1. *stdin*: indica que os dados serão lidos da entrada padrão;
2. uma *string* entre aspas duplas, indicando que os dados serão lidos de um arquivo cujo nome é a *string* especificada;
3. um $fileId$, indicando que os dados serão lidos de um arquivo associado à $fileId$. A associação é feita na linha de comando, ao se chamar o interpretador, da seguinte forma:

$$fileId = fileName$$

Caso a leitura dos arquivos não seja permitida, um erro de execução é gerado. O seguinte exemplo,

input stdin, "in1.txt", in2

mostra que o interpretador manipulará três *strings* de entrada: a primeira é lida da entrada padrão, a segunda do arquivo **in1.txt** e a terceira do arquivo associado a **in2** na linha de comando, quando o interpretador for chamado.

Notus disponibiliza funções pré-definidas para manipulação dessas *strings* de entrada. Elas são listadas na tabela abaixo:

Tabela 2: *Funções pré-definidas em Notus*

trim	$String \rightarrow String$	Ignora caracteres em branco no início da seqüência
nextInt	$String \rightarrow Int$	Retorna o próximo inteiro da seqüência
ignoreInt	$String \rightarrow String$	Ignora caracteres em branco no início da seqüência e o primeiro inteiro no início da seqüência
readLine	$String \rightarrow String$	Retorna todos os caracteres da primeira linha da seqüência
ignoreLine	$StringInt$	Ignora a primeira linha da seqüência

Caso o módulo principal não defina a cláusula *input*, considera-se que o interpretador não manipula sequências de entrada ou manipula somente a entrada padrão, de acordo com a assinatura da função semântica que inicia a avaliação da Árvore Sintática Abstrata gerada pelo analisador sintático.

As sequências de saída são definidas por meio da cláusula, opcional, *output* $out_1, out_2, ..., out_n$, em que cada out_i pode ser:

1. *stdout*: indica que os dados serão escritos na saída padrão;
2. uma *string* entre aspas duplas, indicando que os dados serão escritos em um arquivo cujo nome é a *string* especificada, sobrescrevendo o conteúdo do arquivo, caso ele já exista;
3. **append fileName**, em que **fileName** é uma *string* entre aspas duplas, indicando que os dados serão escritos ao final de um arquivo cujo nome é a *string* especificada;
4. um $fileId$, indicando que os dados serão escritos em um arquivo associado à $fileId$, sobrescrevendo o conteúdo do arquivo, caso ele já exista. A associação é feita na linha de comando, ao se chamar o interpretador, da seguinte forma:

$$fileId = fileName$$

5. **append fileId**, indicando que os dados serão escritos em um arquivo associado à $fileId$, adicionando o conteúdo ao final do arquivo, caso ele já exista. A associação é feita na linha de comando, ao se chamar o interpretador, da seguinte forma:

$$fileId = fileName$$

Arquivos não existentes são criados durante a interpretação. O seguinte exemplo,

output stdout, append "out1.txt", out2

mostra que o interpretador manipulará três seqüências de saída: a primeira é escrita na saída padrão, a segunda no arquivo **out1.txt** e a terceira no arquivo associado a **out2** na linha de comando, quando o interpretador for chamado.

Na ausência da cláusula **output**, uma advertência é gerada e todas as seqüências são escritas na saída padrão.

A função semântica que inicia a avaliação da *AST* é definida por meio da cláusula *semantics f*, onde f pode assumir um dos seguintes tipos:

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

O primeiro parâmetro de f, S, deve ser o domínio do símbolo inicial da gramática. Os demais parâmetros dependem das cláusulas *input* e *output*. Por exemplo, se a função semântica pertencer ao caso 1 anterior e for definida uma seqüência de entrada no módulo principal, uma advertência é emitida durante a compilação e a seqüência é ignorada. Um erro de execução ocorre quando as cláusulas *input* e *output* utilizam identificadores de arquivos e a associação não é feita no momento da chamada ao interpretador.

Segue um exemplo de um módulo *Main*:

```
module Main
   import Grammar;
   syntax program; //Simbolo Inicial
   semantics dmain; //Funcao Semantica Inicial
   output stdout; //Dados de saida sao escritos na saida padrao
```

```
   function dmain : Program -> String -> String;
   dmain program inp = "Hello World!";

end
```

Nesse exemplo, o símbolo inicial da gramática pertence ao domínio **Program** e foi definido no módulo **Grammar**, importado pelo módulo **Main**. Como a cláusula **input** não foi definida no **Main** e a função semântica inicial, **dmain** que é do tipo $Program \rightarrow String \rightarrow String$, exige uma *string* inicial, considera-se que ela será lida da entrada padrão. A *string* de saída será escrita na saída padrão. O exemplo acima simplesmente imprime na saída padrão **Hello World!**.

Segue a definição do módulo **Grammar**, usado no exemplo anterior:

```
module Grammar
 public program ::= "programa";
end
```

4.2 Pacotes

Os módulos de uma definição em *Notus* podem ser organizados em pacotes. Um pacote pode ser entendido como um diretório em um sistema de arquivos, pois pode agrupar outros pacotes (diretórios) e módulos (arquivos). Um módulo M pertence ao pacote P se:

- o cabeçalho do módulo M é **Module P.M**;
- o arquivo $M.nts$ está dentro do diretório P.

Pacotes podem também conter outros pacotes. Se os pacotes P_1, P_2,...,P_n estão aninhados de forma que o pacote P_n é subpacote de P_{n-1}, o módulo M pertence ao pacote P_n se:

- o cabeçalho do módulo M é **Module** $\boldsymbol{P_1.P_2....P_n.M}$;
- se existem os diretórios aninhados P_1, P_2, ..., P_n;
- o arquivo $M.nts$ está dentro do diretório P_n.

5 Especificação Sintática de Linguagens de Programação

5.1 Definição Léxica

A definição dos componentes léxicos em *Notus* é feita por meio de *tokens* e *elements*. Os *tokens* podem ser utilizados na definição sintática, enquanto os *elements* apenas representam partes complexas da definição dos *tokens*, contribuindo para a melhoria da legibilidade. Os *elements* também não podem ser recursivos.

Tokens e *elements* são definidos por expressões regulares e podem possuir controle de visibilidade pública ou privada. As expressões regulares em *Notus* são formadas por caracteres, *Strings*, identificadores e operações: concatenação, feita pelas expressões de justa-posição; união, feita pelo operador |; estrela de *Kleene* feita pelo operador $*$, que representa um número de ocorrências de uma expressão regular; e opção, feita pelo operador ?, representando zero ou uma ocorrência de uma expressão regular. Além disso, expressões podem ser definidas por conjuntos de caracteres enumerados entre colchetes, como por exemplo [**abc**]. É também possível definir conjuntos de caracteres usando intervalos, designando o primeiro e o último elemento do conjunto, como por exemplo [**a-z**]. O símbolo . significa algum caractere exceto nova linha. Também pode-se definir conjuntos de caracteres usando complemento, como o conjunto de todos caracteres exceto a, b ou c, em

```
[^abc].
```

Na definição de conjunto de caracteres, a barra invertida é usada como *escape* para os seguintes símbolos:

```
\n, \r, \f, \b, \", \', \-, \^, \[, \], \\.
```

Concatenação e união associam-se à esquerda, enquanto as outras operações não são associativas. A ordem de precedência é dada na tabela 3:

Seguem alguns exemplos de expressões regulares válidas em *Notus*:

```
"//" .*                                    Comentários em Java
```

Tabela 3: *Precedência de Operadores de Expressões Regulares*

Maior precedência	"*", "+", "?"
	concatenação
Menor precedência	"|"

`"/*" (. | "\n")* "*/"`	`Comentários em C`
`[0-9]+ ("."[0-9]+)? ([eE] "-"? [0-9]+) ?`	`Números de ponto flutuante em C`
`[a-zA-Z_][a-zA-Z_0-9]*`	`Identificadores em C`
`[+\-*/%]`	`Operações aritméticas em C`
`[\n\f\r\t\a\b\"\']`	`Caracteres de escape em C`

Os *tokens* podem pertencer a domínios sintáticos. Na descrição de um *token* também é possível determinar como um lexema será interpretado, especificando-se uma função, a ser aplicada ao lexema quando ele for reconhecido pelo analisador léxico, precedida da cláusula **is**. Essa função somente se aplica aos *tokens* com domínios sintáticos definidos. Ela recebe como argumento uma *String* e retorna um valor do seu domínio sintático do *token*. As seguintes funções são pré-definidas e podem ser usadas para esse fim:

1. $asBoolean : String \rightarrow Bool$
2. $asByte : String \rightarrow Byte$
3. $asShort : String \rightarrow Short$
4. $asInteger : String \rightarrow Integer$
5. $asLong : String \rightarrow Long$
6. $asFloat : String \rightarrow Float$
7. $asDouble : String \rightarrow Double$
8. $asCharacter : String \rightarrow Char$

Quando a cláusula **is** não é utilizada, o lexema do token é tratado como uma *String*.

Exemplos da especificação léxica de uma linguagem em *Notus*:

```
public token id = seqId;
public token num: Num = digit+ is asInteger ;
private element seqId = letter (letter | digit) * ;
public element letter = [A-Z] | [a-z] ;
public element digit = [0-9] ;
```

Os números são definidos como uma seqüência de dígitos, e identificadores são representados por seqüências de letras e dígitos, iniciadas por letras. Os *tokens* **id** e **num** e os *elements* **letter** e **digit** são visíveis para todos os módulos da especificação, enquanto o *element* **seqId** só pode ser usado no módulo onde é definido. A função **asInteger** presente na definição do *token* **num** indica que o lexema do *token* será tratado como inteiro.

A especificação léxica também pode ser feita por meio da cláusula **ignore** que permite ignorar símbolos, como espaços em branco. O seguinte exemplo 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 "//":

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

5.2 Especificação da Gramática Concreta

Em *Notus* a definição da gramática concreta de uma linguagem é feita usando gramáticas livres de contexto LALR(1). A definição das variáveis da gramática concreta de uma linguagem é feita por meio de declarações de variáveis que, de forma semelhante aos *tokens*, podem possuir visibilidade pública, privada ou de pacote. Para a visibilidade pública usa-se a palavra reservada *public*, para a visibilidade privada a palavra *private* e para a visibilidade de pacote não se usa nenhuma palavra chave. A declaração de uma variável não requer palavra reservada, para evitar uma escrita verbosa. Segue um exemplo de produção para uma linguagem aritmética:

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

Nesse exemplo, a variável **exp** possui visibilidade pública, a variável **factor** privada e a **term** possui visibilidade só dentro do pacote. Os *tokens* **num** e **id** usados nesse exemplo são aqueles definidos na seção anterior. Símbolos entre aspas duplas no lado direito de uma regra são tokens anônimos automaticamente incluídos na definição léxica da linguagem. Portanto, no exemplo anterior, os símbolos "+", -", "*", "/", "(", ")"são automaticamente adicionados na definição léxica da linguagem e dispensam declaração explícita para representá-los. Para especificar a gramática concreta pode-se também fazer uso do terminal **empty**, que representa uma *String* vazia, permitindo escrever a regra de produção da forma $A \rightarrow \lambda$.

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. Para permitir repetições dos constituintes das regras do lado direito da produção, podem ser utilizados os símbolos "*", "+", "*-", "+-"após variáveis (não se deve utilizar esses símbolos após *tokens* anônimos), como mostra o exemplo abaixo:

```
programa ::= declaracoes comandos ;
declaracoes ::= declaracao *-";" ;
declaracao ::= "var" id+-"," ;
comandos ::= comando+ ;
comando ::=  id "<-" exp
         |   "if" exp "then" "else" comando
         |    block ;
block ::= "{" comando* "}" ;
```

Nesse exemplo, a variável **declaracoes** produz uma sequência de declarações, que pode ser vazia, separadas por ponto-e-vírgula. A variável **declaracao** produz uma sequência não vazia de identificadores separados por vírgula e precedida da palavra **var**. A variável **comandos** produz uma sequência não vazia de comandos. Por fim, a variável **block** produz uma sequência, que pode ser vazia, de comandos entre chaves.

6 Especificação Semântica de Linguagens de Programação

6.1 Especificação da Gramática Abstrata

Com base nos domínios sintáticos definidos e na gramática concreta da linguagem, o compilador *Notus* gera a gramática abstrata da linguagem que é composta pelos domínios sintáticos dos constituintes das regras, exceto quando o constituinte é um *token* anônimo (nesse caso o constituinte abstrato é o próprio *token*). O exemplo seguinte ilustra a gramática abstrata que é gerada pelo compilador *Notus* com base na respectiva gramática concreta:

```
Gramática Concreta

public exp : Exp ::= exp "+" term   //regra 1
                   | exp "-" term   //regra 2
                   | term ;         //regra 3

term : Exp ::= term "*" primary     //regra 4
           |  term "/" primary     //regra 5
           |   primary ;           //regra 6

primary : Exp ::= id                //regra 7
              | num                 //regra 8
              | "(" exp ")" ;       //regra 9


Gramática Abstrata Gerada

Exp -> Exp "+" Exp
```

```
|  Exp "-" Exp
|  Exp "*" Exp
|  Exp "/" Exp
|  "(" Exp ")"
| Id
| Num

Id -> id
Num -> num
```

É interessante notar que a gramática abstrata é construída com base nos domínios sintáticos dos *tokens* e variáveis das produções da gramática concreta. Quando existe, na gramática concreta, regras do tipo $A \rightarrow A$, sendo A um domínio sintático, o compilador automaticamente simplifica a gramática abstrata. No exemplo anterior, pela regra 3 a seguinte regra $Exp \rightarrow Exp$ apareceria na gramática abstrata, dado que o domínio sintático da variável **term** é **Exp**. O compilador, entretanto, substitui o lado direito de $Exp \rightarrow Exp$ pelo lado direito na definição da variável **term**.

Se uma diferente gramática abstrata é desejada, pode-se fazer uso de uma regra de definição da gramática abstrata para cada produção da gramática concreta, usando o operador ":". 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.

Segue um exemplo de definição alternativa da sintaxe abstrata:

```
Gramática Concreta

public exp : Exp ::= exp "+" term : ["add" exp term]    //regra 1
                 | exp "-" term : ["sub" exp term]      //regra 2
                 | term ;                               //regra 3

term : Exp ::= term "*" primary : ["mult" term primary] //regra 4
           |  term "/" primary : ["div" term primary]  //regra 5
           |   primary;                                 //regra 6

primary : Exp ::= id : id                               //regra 7
              | num : num                               //regra 8
              | "(" exp ")" : exp ;                     //regra 9
```

```
Gramática Abstrata Gerada


Exp -> "add" Exp Exp
|  "sub" Exp Exp
|  "mult" Exp Exp
|  "div" Exp Exp
|  "(" Exp ")"
|  id
|  num


Id -> id
Num -> num
```

Em *Notus* existem dois mecanismos para definir uma regra abstrata. O primeiro é a definição de uma regra da gramática abstrata à direita da regra da gramática concreta entre colchetes. Nesse caso o nodo da árvore sintática abstrata que seria gerado automaticamente pelo compilador é substituído pelo novo nodo especificado. As regras 1, 2, 4 e 5 no exemplo anterior mostram o uso dessa técnica.

O segundo mecanismo é a definição de regras para ignorar a indireção de uma produção, em que se especifica à direita da regra da gramática concreta uma variável ou *token* sem uso de colchetes. Nesse caso cria-se uma ligação direta entre a variável sendo definida e o símbolo da indireção. As regras 7, 8 e 9 no exemplo anterior mostram o uso dessa técnica. A regra 7 inclui a regra abstrata $Exp \rightarrow id$, já que existe a regra $Id \rightarrow id$. De forma semelhante, a regra 8 adiciona a regra abstrata $Exp \rightarrow num$.

As regras 3 e 6 não criam regras alternativas, sendo o compilador responsável por criá-las.

6.2 Funções Semânticas

Para definir uma função semântica é preciso declarar sua assinatura e então declarar o corpo da função, escrevendo pelo menos uma equação de definição. 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. As funções podem também ser declaradas com controle de visibilidade usando as palavras reservadas *public* e *private*.

Uma declaração de função consiste em definir 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, conforme seção 2.1. O exemplo a seguir mostra a declaração de uma função **soma** que deve receber dois inteiros e produzir um terceiro inteiro.

```
public function soma : Int -> Int -> Int ;
```

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. Os parâmetros das definições de funções são padrões a serem casados como os argumentos de uma expressão de aplicação de função. Os nomes dos identificadores usados nos parâmetros da cláusula de definição devem ser distintos.

O exemplo seguinte mostra a definição da função **soma**, declarada anteriormente:

```
soma int1 int2 = int1 + int2;
```

Notus prevê um conjunto de funções pré-definidas, prontas para uso. Elas são listadas na tabela abaixo:

Tabela 4: *Funções pré-definidas em Notus*

acos	$Real \rightarrow Real$	inverso do cosseno de um número entre [-1,1]
asin	$Real \rightarrow Real$	inverso do seno de um número entre [-1,1]
atan	$Real \rightarrow Real$	inverso da tangente de um número
ceil	$Real \rightarrow Real$	arredondamento em direção ao infinito
cos	$Real \rightarrow Real$	cosseno de um ângulo em radianos
cosh	$Real \rightarrow Real$	cosseno hiperbólico de um número
empty	$a* \rightarrow Bool$	verifica se lista está vazia
exp	$Real \rightarrow Real$	exponencial de um número
fix	$(a \rightarrow a) \rightarrow a$	menor ponto fixo de uma função
floor	$Real \rightarrow Real$	arredondamento em direção ao infinito negativo
hd	$a* \rightarrow a$	primeiro elemento de uma lista não-vazia
log	$Real \rightarrow Real$	logarítmo natural de um número positivo
max	$Real* \rightarrow Real$ $Int* \rightarrow Int$	o maior elemento de uma lista não-vazia
min	$Real \rightarrow Real$ $Int* \rightarrow Int$	o menor elemento de uma lista não-vazia
pow	$RealxReal \rightarrow Real$	função potência
round	$Real \rightarrow Real$	arredondamento para o inteiro mais próximo
sqr	$Real \rightarrow Real$	potência quadrada de um número
sqrt	$Real \rightarrow Real$	raiz quadrada de um número
sin	$Real \rightarrow Real$	seno de um ângulo em radianos
sinh	$Real \rightarrow Real$	seno hiperbólico de um número
tan	$Real \rightarrow Real$	tangente de um ângulo em radianos
tanh	$Real \rightarrow Real$	tangente hiperbólica de um número
tl	$a* \rightarrow a*$	cauda de uma lista não-vazia
trunc	$Real \rightarrow Real$	arredondamento em direção a zero

6.3 Padrões

Padrões são utilizados para realizar casamento de padrão, em que o valor de uma expressão é comparado à estrutura do padrão. Eles são utilizados na definição de funções (como parâmetros a serem casados com os argumentos de uma expressão de aplicação de função), em expressões de casamento de padrão (para inspeção de tipo de uma expressão) e em 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*.

Os padrões literais casam apenas com seus respectivos valores, como é o caso do padrão 0 na primeira definição de **fact** do exemplo abaixo:

```
function fact : Int -> Int ;
fact 0 = 1 ;
fact n = n * fact (n-1) ;
```

Os padrões identificadores podem casar com quaisquer valores respeitando as seguintes variações:

- *elementos de enumeração*, que casam apenas com o elemento da enumeração a que ser 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, e casam apenas com valores de um domínio dependente do contexto em que se encontram;
- *identificadores anônimos*, representados pelo caractere _, que casam com qualquer valor. Esses padrões não são associados ao valor casado. Ele é usado para definir o comportamento *default* na definição de funções.

O exemplo anterior, na definição da função **fact**, mostra o uso do padrão com identificador não-declarado através do **n**.

O exemplo seguinte exemplifica o uso de identificadores elementos de enumeração como o padrão **error**, o uso de identificadores decorados de elementos de domínio, com os padrões **int1** e **int2**, e o uso do identificador anônimo _.

```
Value = Int | {error}
function sum : Value -> Value -> Value ;
sum int1 int2 = int1 + int2;
```

```
sum error _ = error ;
sum _ error = error ;
sum _ _ = error ;
```

O padrão agregado tupla casa com valores de um domínio tupla e é representado pelo construtor da tupla seguido pelos seus elementos entre parênteses e separados por vírgula. O exemplo seguinte exemplifica a utilização de padrões de tupla em definições de função. O domínio **P** representa um par ordenado com coordenadas inteiras. A função **quadrante** determina qual é o quadrante do plano cartesiano em que se encontra um determinado ponto **P**.

```
P = (Int, Int) ;
function quadrante : P -> String ;
quadrante P(0,0) = "origem" ;
quadrante P(x,y) = if (x>0 and y>0)
                   then "quadrante 2"
                   else if (x<0 and y<0)
                        then "quandrante 3"
                        else if (x>0 and y<0)
                             then "quadrante 2"
                             else "quadrante 4" ;
```

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 de lista.

No casamento seqüencial os elementos são listados entre parênteses e separados por vírgula. No seguinte exemplo, a função **listP** mapeia uma lista de inteiros em uma nova lista de inteiros de acordo com os elementos da lista de argumentos. De foma semelhante, a função **listP2** mapeia qualquer lista no formato ((1),(),(x,y)) em uma lista (1,x+y).

```
function listP : Int* -> Int* ;
listP () = () ;  // Se lista é vazia, retorna lista vazia
listP (1,2,3) = (1,2,5) ;
listP (1,x,3) = (1,x,4) ;
listP (1,2,_,2) = (1,2,2) ;

function listP2 : Int** -> Int* ;
listP2 ((1), (), (x,y)) = (1,x+y) ;
```

O casamento com identificadores de lista decorados ocorre quando um identificador é decorado pelos símbolos * ou +. O exemplo seguinte mostra a utilização do padrão lista decorado **bool*** na definição da função **listP3** para representar uma lista de valores booleanos que, quando casada, gera a lista **(true,true)** como resultado:

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

No casamento de cabeça/calda, uma lista é descrita por um padrão para o seu primeiro elemento (cabeça) e outro para os demais elementos (cauda), separados por dois pontos. O exemplo a seguir mostra o uso desse padrão na definição da função **lispP4**:

```
function listP4 : Int* -> Int* ;
listP4 () = () ;
listP4 3:s* = 4 : listP4 s* ;
listP4 1:2:s* = 1:2: listP4 s* ;
listP4 int:int1* = int * int : listP4 int1* ;
```

A primeira definição apenas mapeia uma lista vazia em outra. Na segunda definição casam-se argumentos de tipo lista cujo primeiro elemento é o valor inteiro 3.

O padrão nodo *AST* corresponde ao lado direito de uma regra da gramática abstrata existente. Esse padrão é definido entre colchetes, em que podem aparecer apenas padrões de identificadores, *Strings* entre aspas e padrão identificador de lista. Por exemplo, considerando a gramática abstrata abaixo:

```
Exp -> Exp "+" Exp
     | Exp "*" Exp
     | Int
```

A seguinte especificação ilustra o uso desse padrão para definir a semântica de expressões:

```
token num : Exp = [0-9]+ is asInteger ;

exp ::= exp "+" term | term : term ;
term : Exp ::= term "*" factor | factor : factor ;
factor : Exp ::= num | "(" exp ")" : exp ;

function dExp : Exp -> Int ;
```

```
dExp int = int ;
dExp [exp1 "+" exp2] = dExp exp1 + dExp exp2 ;
dExp [exp1 "*" exp2] = dExp exp1 * dExp exp2 ;
```

A primeira definição da função **dExp** determina o resultado da função para um argumento inteiro, a segunda para uma expressão de adição e a terceira para uma expressão de multiplicação.

6.4 Expressões

As expressões em *Notus* podem ser:

1. valores literais;
2. agregados;
3. identificadores;
4. combinação de expressões via operadores unários e binários;
5. aplicação de funções;
6. expressões de padrão;
7. abstração lambda;
8. expressões *let* e *where*;
9. expressões de atualização de função;
10. condicionais;
11. contrução de nó da árvore abstrata.

Valores literais são definidos para cada domínio pré-definido na seção 3.1. Eles podem ser valores booleanos, **true** e **false**; números inteiros, em formato decimal, octal ou hexadecimal; números reais; caracteres; e *strings*.

Em *Notus* existem expressões de agregado para tuplas e listas. O agregado tupla é formado por um construtor e uma seqüência de expressões, separadas por vírgula, e entre parênteses. Seguem exemplos de agregados tupla, em que A, B, C, D e E são construtores.

```
A() B(true) C(1, true, 3 + 7)
D(false,3.1415,E(a,if b then 10 else 9))
```

O agregado lista é uma seqüência de expressões entre parênteses e separadas por vírgula. Seguem exemplos de agregados lista:

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

Identificadores em *Notus* são seqüências de letras, dígitos e o caractere "_", iniciadas com letra minúscula. Identificadores podem ser qualificados com o nome do módulo em que são 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. Segue um exemplo onde se faz uso do identificador **bool1**:

```
Value = Int | Bool ;

function intToBool : Int -> Bool ;
intToBool int1 = bool1
                 where {
                  bool1 = if int1=1
                          then true
                          else false
                 }
};
```

A expressão condicional **if** e a expressão **where**, para introdução de definições locais, utilizadas nesse exemplo são detalhadamente descritas posteriormente, nessa mesma seção.

Notus possui operadores básicos para realização de operações aritméticas, *bit-a-bit*, relacionais, booleanas e manipulação de listas. As tabelas 5, 6, 7, 8, 9, 10 e 11 listam esses operadores. Em cada tabela, a primeira coluna lista os operadores, a segunda o uso de cada operador, a terceira a precedência, a quarta a associatividade e a quinta seus domínios. A precedência é dada por um número, sendo que 1 representa a maior precedência. Precedência 1 é dada para expressões que não envolvem operadores.

Seguem vários exemplos que mostram o uso desses operadores:

- A função **lista1** adiciona um elemento inteiro a uma lista de inteiros, usando o operador de construção de lista;

  ```
  function lista1 : Int* -> Int -> Int* ;
  lista1 int int1* = int : int* ;
  ```

- A função **lista2** usa o operador de concatenação para unir os elementos de duas listas de inteiros;

Tabela 5: *Operadores Aritméticos*

Operador	**Uso**	**Prec.**	**Assoc.**	**Assinatura**
- unário	negação aritmética	2	-	$Int \rightarrow Int$ $Real \rightarrow Real$
*	multiplicação	3	left	$IntxInt \rightarrow Int$ $RealxReal \rightarrow Real$
/	divisão	3	left	$IntxInt \rightarrow Int$ $RealxReal \rightarrow Real$
mod	resto da divisão	3	left	$IntxInt \rightarrow Int$
+	adição	4	left	$IntxInt \rightarrow Int$ $RealxReal \rightarrow Real$
-	subtração	4	left	$IntxInt \rightarrow Int$ $RealxReal \rightarrow Real$

Tabela 6: *Operadores bit-a-bit*

Operador	**Uso**	**Prec.**	**Assoc.**	**Assinatura**
$\sim$	negação *bit-a-bit*	2	-	$Int \rightarrow Int$
>>	shift aritmético para a direita	3	left	$IntxInt \rightarrow Int$
>>>	shift lógico para a direita	3	left	$IntxInt \rightarrow Int$
<<	shift para a esquerda	3	left	$IntxInt \rightarrow Int$
&	conjunção *bit-a-bit*	3	left	$IntxInt \rightarrow Int$
|	disjunção *bit-a-bit*	4	left	$IntxInt \rightarrow Int$
$\wedge$	*or* exclusivo *bit-a-bit*	4	left	$IntxInt \rightarrow Int$

Tabela 7: *Operadores Relacionais*

Operador	**Uso**	**Prec.**	**Assoc.**	**Assinatura**
<	menor que	5	-	$RealxReal \rightarrow Bool$ $IntxInt \rightarrow Bool$
>	maior que	5	-	$RealxReal \rightarrow Bool$ $IntxInt \rightarrow Bool$
<=	menor ou igual a	5	-	$RealxReal \rightarrow Bool$ $IntxInt \rightarrow Bool$
>=	maior ou igual a	5	-	$RealxReal \rightarrow Bool$ $IntxInt \rightarrow Bool$
=	igual a	5	-	$Int \rightarrow Int \rightarrow Int$
!=	diferente de	5	-	$RealxReal \rightarrow Bool$ $IntxInt \rightarrow Bool$

Tabela 8: *Operadores Booleanos*

Operador	**Uso**	**Prec.**	**Assoc.**	**Assinatura**
not	negação	2	-	$Bool \rightarrow Bool$
and	conjunção	6	left	$BoolxBool \rightarrow Bool$
or	disjunção	7	left	$BoolxBool \rightarrow Bool$

Tabela 9: *Operadores de Listas*

Operador	**Uso**	**Prec.**	**Assoc.**	**Assinatura**
:	construção de lista	8	right	$axa* \rightarrow a*$
++	concatenação de lista	9	left	$a*xa* \rightarrow a*$

```
function lista2 : Bool* -> Bool* -> Bool* ;
lista2 bool1* bool2* = bool1* ++ bool2* ;
```

- A função **checkValue** faz uso do operador de casamento de padrão, *is*, para verificar se um valor é inteiro;

```
Value = Int | {error} ;

function checkValue : Value -> String ;
checkValue value = if value is int
                   then "Inteiro"
                   else "Erro" ;
```

- A função **intToString** transforma um inteiro em uma *String*, com auxílio da função **digitToInt** e de operadores aritméticos;

```
function intToString : Int -> String ;
intToString int = case int/10 of {
                   0 -> digitToString int ;
                   _ -> intToString (int / 10) ++ intToString (int mod 10)
                  } ;
```

Tabela 10: *Operadores de Composição de Funções*

Operador	**Uso**	**Prec.**	**Assoc.**	**Assinatura**
.	função de composição	10	right	$(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$

Tabela 11: *Operadores de Casamento de Padrão*

Operador	**Uso**	**Prec.**	**Assoc.**	**Assinatura**
is	casamento de padrão	11	-	$a \rightarrow b \rightarrow Bool$

```
function digitToString : Int -> String ;
digitToString int = case int of {
                    0 -> "0" ;
                    1 -> "1" ;
                    2 -> "2" ;
                    3 -> "3" ;
                    4 -> "4" ;
                    5 -> "5" ;
                    6 -> "6" ;
                    7 -> "7" ;
                    8 -> "8" ;
                    9 -> "9"
                    }  ;
```

- A função **nand** realiza a operação booleana *nand*, dado dois booleanos;

```
function nand : Bool -> Bool -> Bool ;
nand bool1 bool2 = not (bool1 and bool2) ;
```

- A função **multiplica** realiza a multiplicação de um inteiro por 2 usando um operador *bit-a-bit*;

```
function multiplica : Int -> Int ;
multiplica int = int << 1 ;
```

- A função **fact** calcula o fatorial de um número e a função **fact1**, por fim, utiliza o operador de composição para calcular o fatorial de um número e, em seguida, transformar o resultado em uma sequência de caracteres.

```
function fact : Int -> Int ;
fact 0 = 1 ;
fact n = n * fact (n-1)

function fact1 : Int -> String ;
fact1 int = (intToString . fact) int ;
```

Aplicações de funções possuem a forma:

$$e_0\, e_1\, e_2 \ldots e_n,$$

em que cada e_i é uma expressão. Se e_i possui domínio d_i e e_0 possui domínio $d_1 \rightarrow d_2 \rightarrow .. \rightarrow d_n \rightarrow d$, então a aplicação de função $e_0\, e_1\, e_2 \ldots e_n$ 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 $d_1 \rightarrow d_2 \rightarrow \ldots \rightarrow d_n \rightarrow d$, com corpo composto por m definições de funções, cada uma delas composta pelo conjunto de padrões. A avaliação dos argumentos da aplicação de função $f\, e_1\, e_2 \ldots e_n$ 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 os argumentos da aplicação forem complexos, pode-se fazer uso de parênteses. A fim de melhorar a legibilidade, a cláusula **evaluate** de *Notus* pode ser empregada:

```
evaluate { functionName ; arg1 ; arg2 ; ... ; argn }
```

que é equivalente à:

```
functionName ( arg1 ( arg2 ( ... ( argn ) ) ) )
```

Por exemplo, a expressão

```
evaluate {fun a b; c d; e f}
```

é equivalente a

```
fun a b (c d (e f))
```

A expressão de casamento de padrão executa apenas uma inspeção de tipo e, portanto, não associa o valor da expressão ao padrão testado. O padrão casado, portanto, não pode ser usado fora da expressão de casamento de padrão. Essa expressão tem a forma **e is p**, em que e é uma expressão cujo domínio será casado com o padrão p. O resultado dessa expressão é um valor booleano. Segue um exemplo dessa expressão:

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

function pExp : Value -> String ;
pExp value = if value is int then "int value"
             else if value is bool then "bool value"
             else "error" ;
```

Nesse exemplo, **pExp** utiliza a expressão de casamento de padrão para testar se o argumento **value** pertence a um dos domínios **Int**, **Bool** ou **error**.

Abstrações lambda têm a forma $\backslash p_1\, p_2 \ldots p_n \rightarrow e$, em que cada p_i é um padrão e e uma expressão. Segue um exemplo dessa expressão:

```
function subtracao : Int -> Int -> Int ;
subtracao = \int1 int2 -> int1 - int2 ;
```

Nesse exemplo, a definição da função **subtracao** retorna uma função, utilizando a abstração lambda, que subtrai um inteiro de outro.

Expressões *let* e *where* são utilizadas para introduzir definições locais auxiliares. A expressão *let* tem a forma

$$let\,\{p_1 = e_1;\; p_2 = e_2;\; \ldots;\; p_n = e_n\}\,in\,e$$

e a expressão *where* tem a forma

$$e\,where\,\{p_1 = e_1;\; p_2 = e_2;\; \ldots;\; p_n = e_n\},$$

em que cada p_i é um padrão, cada e_i uma expressão e e uma expressão. O valor da expressão e é avaliado em um ambiente que faz uso dos vários $p_i s$. As sub-expressões podem ser mutuamente recursivas de forma que cada e_i pode referenciar qualquer p_j. Segue um exemplo dessas expressões:

```
function wherefib : Int -> Int ;
wherefib 0 = 0;
wherefib 1 = 1;
wherefib n = x + y where { x= wherefib (n-1); y= wherefib (n-2)};

function wherefib : Int -> Int ;
wherefib 0 = 0;
wherefib 1 = 1;
wherefib n = let { x= wherefib (n-1); y= wherefib (n-2)} in  x + y ;
```

A expressão de atualização de uma função em *Notus* é usada para atualizar determinados pontos de uma função. Ela tem a forma $f[a \leftarrow v]$, em que f é uma função que aplicada ao argumento a retorna o novo valor v. O seguinte exemplo ilustra a aplicação dessa expressão na definição da função **g**:

```
function fact : Int -> Int ;
fact 0 = 1 ;
```

```
fact n = n * fact (n - 1);


function g : Int -> Int ;
g = fact[1<-5];
```

Assim, g é definida como:

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

A expressão *if* possui a forma

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

em que e_1 é uma expressão do tipo booleano e e_2 e e_3 são expressões que possuem tipos compatíveis. O seguinte exemplo mostra o uso da expressão *if*:

```
function operacaoOr : Bool -> Bool -> Bool ;
operacaoOr bool1 bool2 = if bool1
                         then true
                         else if bool2
                              then true
                              else false ;
```

A expressão *case* possui a forma

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

em que e_i é uma expressão e p_i um padrão. O valor da expressão *case* é e_i se o resultado da avaliação de e_0 casar com o padrão p_i. O seguinte exemplo mostra o uso da expressão *case*:

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


function caseExp : Value -> String ;
caseExp value = case value of {
                     int  -> "int value" ;
                     bool -> "bool value" ;
                     _    -> "error"
                };
```

Cada expressão em *Notus* possui um tipo associado, ou seja, um domínio associado. A verificação de tipos consiste em avaliar o tipo de cada expressão, de acordo com o tipo esperado para ela. A compatibilidade entre dois tipos t_1 e t_2 ocorre quando o tipo t_2 pode ser usado quando o tipo t_1 é esperado. Isso acontece se t_2=t_1 ou se t_2 é subtipo de t_1.

Em *Notus* também é possível construir uma expressão que é um nodo da árvore sintática abstrata. Essas expressões são úteis, por exemplo, quando a semântica de uma linguagem é dada em várias partes (por exemplo, estática e dinâmica) e a cada parte da semântica são necessárias modificações na árvore sintática abstrata. Como no padrão nodo da AST, a expressão nodo AST é definida entre colchetes, em que podem aparecer apenas expressões de identificadores e *Strings* entre aspas. Por exemplo, considerando a seguinte gramática abstrata de uma pequena linguagem de programação:

```
Exp -> Exp "+" Exp
     | Exp "=" Exp
     | Int

Com -> Com Com
     | "if" Exp "then" Com
     | "while" Exp Com
     | Id "=" Exp

token id : Id = [a-z];
```

O seguinte exemplo ilustra o uso da expressão de nodo *AST*. A primeira função, constrói uma expressão de somas de expressões, dado uma expressão e um valor inteiro. A segunda, transforma os comandos, de forma que o usuário da linguagem possa escrever na expressão de um *if* e *while* apenas um valor inteiro e a checagem estática transforme essa expressão inteira em uma expressão de igualdade que, de fato, será executada.

```
function constroiExp : Exp -> Int -> Exp;
constroiExp exp int = [exp "+" exp1] where { exp1 = int };

function tCom : Com -> Com;
tCom ["if" exp "then" com] = ["if" exp1 "then" com1]
   where { exp1 = [exp "=" exp2];
           exp2 = int;
           int = 1;
           com1 = tCom com
   };
```

```
tCom ["while" exp com] = ["while" exp1 com1]
    where {
           exp1 = [exp "=" exp2];
           exp2 = int; int = 1;
           com1 = tCom com
    };
tCom [com1 com2] = [com3 com4]
    where {
           com3 = tCom com1;
           com4 = tCom com2
    };

tCom [id "=" exp] = [id "=" exp];
```

É importante notar, no exemplo acima, que nenhuma conversão automática de tipo (domínio) é prevista pelo compilador, sendo o usuário de *Notus* responsável por fazer essa conversão explicitamente. Por exemplo, na definição da função **constroiExp** não poderia ser retornado $[exp" + " int]$ diretamente, porque na gramática abstrata não pode haver uma expressão de forma $Exp" + " Int$. Embora uma expressão possa ser Int, essa conversão não é feita automaticamente pelo compilador.

7 Extensões

O principal uso desta característica de *Notus* é para suportar extensão na definição de linguagens de programação, permitindo que novos elementos sejam adicionados sem modificar módulos já existentes em especificações anteriores. Além disso, a inclusão desse recurso em *Notus* é devido a ortogonalidade de linguagens, incluindo a extensão de *tokens*, variáveis de gramáticas, domínios e funções.

7.1 Cláusula Extend

Notus permite a extensão de definições de *tokens* e elementos de *tokens* por meio da união de uma nova expressão regular. Essa extensão é realizada por meio da cláusula *extend*. Para exemplificar, considere o código abaixo contendo a declaração dos *tokens* **num** e **id**:

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

O exemplo seguinte, mostra a extensão dos *tokens* **num** e **id**, fazendo com que a linguagem definida suporte números em hexadecimal e permita que identificadores iniciem com o caractere _.

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

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 forma, quando uma *String s* for reconhecido pela expressão regular $[0-9]+$, a função **getNumber** é aplicada a *s*. Por outro lado, se a *String* for reconhecida pela expressão $"0x"[0-9A-Fa-f]+$, a função **getNumbeInHexa** é a que será aplicada a *s*. Quando nenhuma nova função é especificada, como é o caso da extensão do *token* **id**, se o *String* for reconhecido pela expressão regular $"\_"[a-z]+[0-9]*$ a função **getId** é que será aplicada.

A extensão de elementos de *tokens* é feita de forma semelhante a de *tokens*. Segue um exemplo:

```
element letter = [a-z] ;
extend element letter with [A-Z] ;
```

Inicialmente, *letter* denota apenas letras minúsculas e, após a extensão, denota também letras maiúsculas.

A extensão da gramática concreta, dada pela adição de novas regras de produção a variáveis já definidas, é feita com o uso também da cláusula *extend*. O seguinte exemplo ilustra seu uso:

```
public exp: Exp ::= exp "+" term | term ;
public term: Exp ::= term "*" factor | factor ;
public factor: Exp ::= id | num ;
.
.
.
extend exp with exp "-" term;
extend term with term "/" factor ;
extend factor with "(" exp ")";
```

A primeira parte desse exemplo define apenas as operações de adição e multiplicação. A segunda parte define as operações de subtração, por meio da extensão da variável **exp**, de divisão, por meio da extensão da variável **term**, e de expressões entre parênteses, por meio da extensão da variável **factor**.

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

A extensão de domínios é feita de forma semelhante a extensão de variáveis da gramática, por meio da cláusula *extend*. O exemplo que segue ilustra a extensão do domínio **Language**:

```
Language = English | Portuguese;
Portuguese = {um, dois, tres};
English = {one, two, three};

extend Language with Spanish;
Spanish = {uno, dos, tres};
```

A definição de **Language** anterior resulta em um domínio que é equivalente ao obtido com a seguinte definição:

```
Language = English | Portuguese | Spanish;


Portuguese = {um, dois, tres};
English = {one, two, three};
Spanish = {uno, dos, tres};
```

7.2 Transformações

As transformações de *Notus* são também recursos para possibilitar a escrita incremental de linguagens de programação. Com elas é possível, por exemplo, alterar a declaração de funções, bem como redefiní-la, sem causar grandes impactos em especificações anteriores.

Primeiramente, é importante ressaltar a estrutura necessária dos pacotes e módulos de *Notus* para uso de transformações. Supondo que esteja se fazendo uma definição incremental de uma dada linguagem. A linguagem com construções base é definida usando um projeto *Notus* de nome L_0. Nessa definição inicial, deve haver o módulo *Main*. Posteriormente, decide-se expandir essa linguagem base adicionando a ela novas construções. Então, criam-se vários projetos *Notus*, L_1, L_2, $L_3 \ldots L_n$, em que cada nova definição L_i expande a sub-linguagem defina por L_{i-1}, $i > 0$. As seguinte restrições devem ser observadas:

- no projeto L_i pode haver novos módulos que adicionam novos tokens e variáveis à definição anterior. Entretanto, nenhum módulo de L_i pode ter o mesmo nome completo de outro módulo presente em L_0, ..., L_{i-1} (exceto o módulo **Main.nts**);
- quando em L_i se fizer uso de elementos de L_0, ..., L_{i-1}, através da cláusula *import*, por exemplo, deve-se omitir o nome do projeto onde eles foram definidos e utilizá-los como se tivessem integrados à L_i. Assim, se for necessário utilizar a função **x** definida no módulo **A** de L_0, deve-se utilizar, em L_i apenas $A.x$.
- em L_i deve haver um módulo específico de transformações. Nele serão feitas todas as transformações desejadas das funções de L_0, ..., L_{i-1} usando as cláusulas específicas para transformação que serão apresentadas posteriormente. O módulo de transformação deve ter o cabeçalho da seguinte forma:

  transformation $TransformationName$

- a especificação semântica de L_i já deve considerar que todas as transformações definidas no projeto foram feitas. Assim, se um parâmetro foi adicionado em alguma função de L_0, ..., L_{i-1} através de transformações, deve-se utilizar essa função em L_i considerando esses novos parâmetros;
- deve haver em L_i um módulo **Main** que especifica o módulo de transformações e qual é o projeto a que essas transformações serão aplicadas da seguinte forma, quando se estende a especificação de $L_i, i > 0$:

module $Main$

extension $TransformationName\,(L_i)$

end

Depois de mostrar a estrutura e as restrições a que devem estar sujeitos os vários projetos para definir incrementalmente uma linguagem, apresentam-se as cláusulas de extensão que podem estar contidas nos módulos de transformação presentes em cada L_i, $i > 0$. Para isso, a especificação semântica de uma sub-linguagem L_0 será primeiramente fornecida e, posteriormente, a especificação de uma nova sub-linguagem L_1, que estende L_0 por introduzir o comando *break*, será apresentada.

Segue a gramática abstrata de L_0:

```
Exp -> Exp "+" Exp
     | Exp "=" Exp
     | Int
     | Id

Com -> Com Com
     | "if" Exp "then" Com
     | "while" Exp Com
     | Id "=" Exp

token id : Id = [a-z]

Dec -> Dec Dec
      | Dec Com
      | Type Id

Type -> Int

Prog -> Dec
```

L_0 é uma sub-linguagem bem simples que possui comandos de repetição, atribuição e condicional. As expressões podem ser soma aritmética, comparação, literal e valor de variável. O único tipo possível é inteiro. As declarações podem ser somente de variável e devem ser feitas antes do uso.

Os domínios semânticos usados na especificação de L_0 são definidos no módulo **Domains**, como é mostrado a seguir:

```
module Domains


 Bv    = Int;
 Loc   = Int;
 Dv    = Loc;
 Env   = Id -> EnvValue;
 Store = Loc -> StoreValue;
 StoreValue = Bv | Unused;
 EnvValue = Dv | Unbound;
 Cc    = Store -> A;
 Ec    = Bv -> Store -> A;
 Dc    = Env -> Store -> A;
 A     = AnsValue (Bv,A) | {error,stop} ;
 Unbound = {unbound};
 Unused = {unused};


end
```

Bv representa os tipos básicos da linguagem. **Dv** os valores que podem ser associados às variáveis no *environment*. Esse, associa um identificador de variável à uma posição de memória. A memória **Store** mapeia posições de memória em valores básicos. **Cc**, **Ec** e **Dc** são continuações de comando, expressão e declaração, respectivamente. Continuação é o resto do programa que segue uma dada construção. Para maior entendimento desse conceito ver [2]. A resposta final do programa é uma sequência de valores básicos seguidos por *stop*, se a execução prosseguiu com sucesso, ou *error*, em caso contrário.

As funções utilizadas na especificação, declaradas no módulo **Functions**, seguem:

```
module Functions


import Domains;


function dexp  : Exp -> Env -> Ec -> Store -> A;  //declaracao de funcao
function dcom  : Com -> Env -> Cc -> Store -> A;  //declaracao de funcao
function ddec  : Dec -> Env -> Dc -> Store -> A;  //declaracao de funcao


function update : Env -> Env -> Env;
update r r1 = let {
r2 =  \id ->  if r1 id is unbound
```

```
  then r id
  else r1 id
 in r2;

function newEnv : Env;
newEnv = \id -> unbound;

function new : Store -> Loc;
new store = findNew store 0;

function findNew : Store -> Loc -> Loc;
findNew store int = if (store int) is unused
                    then int
                    else findNew store newLoc
                         where {newLoc = int+1};

end
```

A função **update** une dois *environments*, **new** retorna uma posição livre da memória e **newEnv** retorna um pequeno *environment* que mapeia qualquer identificador para **unbound**. As funções **dexp**,**ddec** e **dcom** descrevem o significado, respectivamente, das expressões, das declarações e dos comandos de L_0.

No módulo **Semantics** estão as várias definições, por casamento de padrão, das equações semânticas **ddec**, **dexp** e **dcom**, declaradas no módulo **Functions**. A semântica de uma construção deve ser sempre baseada na semântica de seus constituintes.

A semântica de uma expressão formada pela combinação de duas sub-expressões, que constituem os operandos de uma operação, é descrever a semântica de cada sub-expressão e aplicar os valores obtidos, ao avaliá-las, ao operador da expressão. O valor e o estado resultantes são passados à continuação de expressão. A semântica de uma expressão que é apenas um identificador consiste em recuperar o valor do identificador do *environment* e da memória de execução e transmití-lo à continuação de expressão. Por fim, a semântica de uma expressão que é um literal inteiro é passar à continuação esse valor.

A semântica de um comando **if** *exp* **then** *com* consiste em avaliar a expressão do comando, que retorna um valor. Caso esse valor retornado seja um, o comando interno é executado, caso contrário, segue-se para a continuação de comando do **if**. A semântica do comando **while** *exp com*, de forma semelhante, executa o comando interno somente se o valor retornado ao avaliar a expressão for um. Nesse caso, a continuação do comando interno é executar novamente o comando **while**. A semântica do comando de atribuição *id*" = " *exp* consiste em atribuir ao identificador, no *environment*, a posição de memória atualizada com o valor retornado ao avaliar a expressão. Por fim, a semântica de *com*1 *com*2 é avaliar o segundo comando em uma memória que é resultante da avaliação

do primeiro comando.

A semântica da declaração de uma variável, *type id*, é reservar uma posição de memória e associá-la ao identificador no *environment*. A semântica de uma declaração seguida de um comando, *dec com*, é atualizar o *environment* com as declarações dos identificadores e executar o comando nesse *environment* atualizado. Por fim, a semântica de duas declarações seguidas é declarar os identificadores da segunda declaração em um *environment* que contém todos os identificadores da primeira declaração.

```
module Semantics

import Domains, Functions;

dexp [exp1 "+" exp2] r k s = dexp exp1 r k1 s
where {
 k1 = \bv s1 -> dexp exp2 r k2 s1
    where {
       k2 = \bv1 s2 -> k (bv + bv2) s2
    }
};

dexp [exp1 "=" exp2] r k s = dexp exp1 r k1 s
where {
 k1 = \bv s1 -> dexp exp2 r k2 s1
    where {
       k2 = \bv1 s2 -> if bv = bv2 then k s2 1
                       else k s2 0
    }
};

dexp int r k s = k int s;

dexp id r k s = if r id is unbound
                then error
                else if s (r id) is unused
                     then error
                     else k (s (r id)) s;
```

```
dcom [com1 com2] r c s =  dcom com1 r c1 s
where {
 c1 = \s1 -> dcom com2 r c s1
};

dcom ["if" exp "then" com] r c s =  dexp exp r k s
where {
 k = \bv s1 -> if bv=1 then dcom com1 r c s1
               else c s1
};

dcom ["while" exp com] r c s =  dexp exp r k s
where {
 k = \bv s1 -> if bv=1 then dcom com r c1 s1
                 where{
                   c1 = \s2 -> dcom ["while" exp com] r c s2
                 }
               else c s1
};

dcom [id "=" exp] r c s =  dexp exp r k s
where {
 k = \bv s1 -> if r id is unbound
               then error
               else c (s[bv<-loc])
                  where {
                    loc = r id
                  }
};

ddec [dec1 dec2] r u s = ddec dec1 r u1 s
where {
 u1 = \r1 s1 -> ddec dec2 r2 u2 s1
       where {
         r2 = update r r1;
```

```
          u2 = \ r3 s2 -> u (update r1 r3) s2
        }
};

ddec [dec com] r u s = ddec dec r u1 s
where {
 u1 = \r1 s1 -> dcom com r2 c s1
        where {
          r2 = update r r1;
          c = \s2 -> u r1 s2
        }
};

ddec [type id] r u s = if r id is unbound
 then u newEnv[loc<-id] s
 where{
                         loc = new s
                        };
                        else error
```

Dada essa especificação feita em L_0, o comando *break* será adicionado à sub-linguagem original, em L_1.

Para inserir o comando **break**, que dentro de um comando **while** provoca a terminação do *loop*, é necessária uma continuação adicional, que corresponde à continuação do comando *while* dentro do qual o *break* está definido.

Uma maneira de tratar a inclusão do *break* é a utilização de um parâmetro adicional na avaliação de cada comando. Esse novo parâmetro é a continuação a ser seguida pelo comando *break*, que ignora a continuação normal do programa.

A primeira cláusula de transformação a ser apresentada é **signature**, que permite adicionar novos parâmetros às funções já definidas em especificações anteriores. Segue o uso de **signature** para efetuar a adição de mais um parâmetro à função **dcom**. Essa transformação é definida no módulo de transformação **Break** (é necessário, por isso, o uso da palavra **transformation** ao invés de **Module**):

```
transformation Break

signature dcom  : Com -> Env -> Cc -> Store -> A to
                  Com -> Env -> Cc -> (newCont: Cc) -> Store -> A;
```

```
end
```

O módulo de transformação pode também importar outros módulos, como mostrado em 4.1. O módulo **Break** de L_1 define a nova assinatura da função **dcom**, adicionando o novo parâmetro nomeado como **newCont**. O tipo do novo parâmetro é especificado como uma continuação de comando (**Cc**). Por *default*, cada equação **dcom** em L_0 que se chama recursivamente, passa como valor para o novo parâmetro, na chamada recursiva, o valor que é recebido como parâmetro. É importante ressaltar que podem ser adicionado vários parâmetro em uma mesma função e também pode-se, usando a mesma cláusula **signature**, especificar adições de parâmetros em várias funções da seguinte forma:

signature

$$function_A : A_1 \rightarrow ... \rightarrow A_n \textbf{ to}$$
$$A_1 \rightarrow ... \rightarrow (new_A : A_i) \rightarrow A_n,$$
$$function_B : B_1 \rightarrow ... \rightarrow B_n \textbf{ to}$$
$$B_1 \rightarrow ... \rightarrow (new_B : B_j) \rightarrow B_n,$$
$$...$$
$$function_N : N_1 \rightarrow ... \rightarrow N_n \textbf{to}$$
$$N_1 \rightarrow ... \rightarrow (new_N : N_k) \rightarrow N_n;$$

Como um programa é uma lista de declarações e essas podem ser declarações seguidas de comando, a semântica de declaração dependerá da semântica de comando. Logo, um primeiro valor para esse novo parâmetro (**newCont**) deverá ser especificado. Isso pode ser definido em L_1 com o uso da cláusula de transformação **default**. No exemplo utilizado, o valor a ser passado para a função semântica, na ausência de outro valor definido, deve ser um erro, pois somente o comando **while** pode definir o valor dessa continuação. Logo, se a continuação adicional for utilizada e não haver comando **while** é porque um **break** foi definido fora de um **while**. Tem-se a seguinte nova definição:

```
transformation Break


signature dcom  : Com -> Env -> Cc -> Store -> A to
                  Com -> Env -> Cc -> (newCont: Cc) -> Store -> A;


default newCont = \s -> error;


end
```

Na cláusula **default** deve-se especificar o label do novo parâmetro definido e o valor por ele assumido. Como pode haver vários novos parâmetros adicionados, inclusive em

várias funções, na mesma cláusula **default** é possível especificar o valor inicial para vários novos argumentos da seguinte forma:

signature

$$function_A : A_1 \rightarrow ... \rightarrow A_n \textbf{ to}$$
$$A_1 \rightarrow ... \rightarrow (new_A : A_i) \rightarrow A_n,$$
$$function_B : B_1 \rightarrow ... \rightarrow B_n \textbf{ to}$$
$$B_1 \rightarrow ... \rightarrow (new_B : B_j) \rightarrow B_n,$$
$$...$$
$$function_N : N_1 \rightarrow ... \rightarrow N_n \textbf{to}$$
$$N1 \rightarrow ... \rightarrow (new_N : N_k) \rightarrow N_n;$$

default $new_A = e_1$, $new_B = e_2, ..., newN = e_n$;

e_1, , e_2, ..., e_n são expressões que devem ter tipos compatíveis com o tipo com o qual o *label* foi definido.

Agora, precisamos redefinir o comando **while** para que ele propague para os comandos internos a ele a nova continuação. Isso pode ser feito por meio da cláusula **redefine**, que permite redefinir o corpo de função, como é mostrado a seguir:

```
transformation Break

signature dcom  : Com -> Env -> Cc -> Store -> A to
                  Com -> Env -> Cc -> (newCont: Cc) -> Store -> A;

default newCont = \s -> error;

redefine dcom ["while" exp com] r c c1 s by
dexp exp r k s
 where {
  k = \bv s1 -> if bv=1 then dcom com r c2 c s1
                 where{
                   c2 = \s2 -> dcom ["while" exp com] r c c1 s2
                 }
                else c s1
};
end
```

Nota-se que o corpo da função **dcom** é redefinido somente quando os argumentos

de **dcom** casam com os que foram especificados na cláusula **redefine**. Assim, **dcom** só é redefinido quando o primeiro argumento, um **Com**, tiver a forma de um comando **while**. De maneira semelhante que as demais cláusulas já apresentadas, pode-se redefinir várias funções usando a mesma cláusula **redefine**, apenas separando cada redefinição por vírgula.

Por fim, a semântica do novo comando de L_1, que deve considerar os transformadores aplicáveis à L_0, é fornecida no módulo **NewComands**. A especificação L_1 deve também ter um módulo **Main**. Nesse caso, como essa definição estende outra anterior, nesse módulo são especificados o módulo em que as claúsulas de transformações estão e qual especificação anterior L_1 estende.

```
transformation Break

signature dcom  : Com -> Env -> Cc -> Store -> A to
                  Com -> Env -> Cc -> (newCont: Cc) -> Store -> A;

default newCont = \s -> error;

redefine dcom ["while" exp com] r c c1 s by
dexp exp r k s
 where {
  k = \bv s1 -> if bv=1 then dcom com r c2 c1 s1
                 where{
                   c2 = \s2 -> dcom ["while" exp com] r c c1 s2
                 }
                else c s1
};
end

module NewCommands

dcom [break] r c1 c2 s =  c2 s;

end

module Main

extension Break(L0)
```

```
end
```

Uma outra maneira de incluir o comando *break*, sem necessidade de alterar a assinatura da função **dcom**, é associar a continuação a ser seguida pelo *break* a uma posição específica do *environment* chamada **break**. Para isso, a cláusula **replace** pode ser utilizada. Essa cláusula possui a seguinte forma:

$$\textbf{replace}\, f\, p_1 \ldots p_k\, [\, \textbf{within}\, g_1\, p'_1 \ldots p'_m\,]\, \textbf{by}\, e_1$$

onde f é uma função e cada p_i um padrão que deve casar com o **i-ésimo** argumento de f. O efeito dessa cláusula é substituir todas as aplicações da função f cujos argumentos casem com os padrões especificados. Pode-se também usar a cláusula **replace** para substituir a aplicação de várias funções, apenas separando cada substituição por vírgula.

Com o uso dessa cláusula o módulo de transformação é:

```
transformation Break

replace dcom com r c1 s1 within dcom ["while" exp com] r c s
by dcom com r[break<-c] c1 s1;

end
```

A semântica completa de L_1, que inclui a denotação do novo comando de L_1, os transformadores e o módulo **Main** é mostrada abaixo:

```
transformation Break

replace dcom com r c1 s1 within dcom ["while" exp com] r c s
by dcom com r[break<-c] c1 s1;

end

module NewCommands

dcom [break] r c1 s =  case (r break) of {
                          c -> c s;
                          _ -> error
                       }
```

```
end

module Main

extension Break(L0)

end
```

8 Executando o Compilador Notus

Para executar o compilador *Notus* são necessários os seguintes programas:

- *Java JRE* versão 5.0 ou superior, disponível em
  $http://java.com/en/download/index.jsp$;
- gerador de analisadores léxicos para *Haskell*, *Alex* versão 2.0.1 ou superior, disponível em $http://www.haskell.org/alex/$;
- gerador de analisadores sintáticos para *Haskell*, *Happy* versão 1.15 ou superior, disponível em
  $http://www.haskell.org/happy/$;
- compilador de *Haskell*, *GHC* versão 6.6 ou superior, disponível em
  $http://www.haskell.org/ghc/$.

É importante assegurar que os caminhos onde estes programas foram instalados estão no *classpath* do sistema.

A versão mais recente do compilador *Notus* pode ser obtida em $www.dcc.ufmg.br/tays$. Ela possui uma pasta *lib* contendo as bibliotecas usadas e o arquivo *jar* executável *notus.jar*. O caminho de todos os arquivos dentro da pasta *lib* (*aspctjrt.jar*, *automaton.jar*, *cup_JLex.jar*), assim como do arquivo *notus.jar*, devem ser colocados no *classpath*.

8.1 Executando o Compilador na Linha de Comando

Para executar uma especificação no compilador *Notus*, é preciso digitar na linha de comando:

```
java notus.Notus <src> <out>
```

onde $< src >$ é a pasta, com seu caminho completo, que contém o módulo *Main* da especificação e $< out >$ é a pasta, com seu caminho completo, onde o interpretador será gerado. Em seguida, deve-se executar os geradores *Alex*, *Happy* e *ghc* para os arquivos resultantes do processo anterior:

```
alex Lexer.x
happy Syntactic.y
ghc --make -o <nome do interpretador> Main
```

O interpretador pode, então, ser executado na linha de comando e o primeiro argumento é o nome do interpretador gerado, seguido, opcionalmente, dos argumentos de entrada e saída, de acordo com a especificação da linguagem.

Por exemplo,

```
alex Lexer.x
happy Syntactic.y
ghc --make -o compiladorJava Main

compiladorJava Programa1
```

*Programa*1 é um programa Java a ser compilado e executado pelo *compiladorJava* gerado por *Notus*.

8.1.1 Integrando o Compilador ao Eclipse

O plug-in descrito em [3] integra o compilador de *Notus* ao aplicativo *Eclipse*.

É preciso copiar o plug-in para a pasta **plugin** do Eclipse. Por exemplo, se o Eclipse foi instalado na pasta $C:\backslash Eclipse$, deve-se copiar o plug-in para $C:\backslash Eclipse\backslash plugins$.

Depois disso ser feito, quando a opção $New > Other...$, no Eclipse, for escolhida, aparecerá uma alternativa para criação de um projeto em *Notus*. Quando um projeto *Notus* é criado, o módulo **Main.nts** é também automaticamente criado na pasta principal do projeto. Novas pastas podem ser inseridas ao projeto, correspondendo a criação de pacotes em *Notus*. De forma semelhante, novos arquivos **nts** podem ser adicionados ao projeto principal ou aos pacotes criados dentro do projeto.

9 Conclusão

A linguagem funcional *Notus* foi apresentada neste manual, que reuniu várias informações da linguagem disponíveis em diversas fontes. Como pôde ser notado, a descrição formal das construções de *Notus* não foi o foco principal deste trabalho. Ao invés disso, as construções da linguagem foram apresentadas através de vários exemplos, visando facilitar o entendimento e aprendizado do usuário. A apresentação incluiu as diversas construções de *Notus* comuns em demais linguagens funcionais e as específicas para definição de semântica formal, bem como os recursos da linguagem para permitir incrementabilidade e modularidade nas definições. Por fim, foram descritos os procedimentos para obter, instalar e utilizar o compilador de *Notus*, inclusive como integrá-lo à ferramenta *eclipse*.

Referências

[1] Tays Cristina do Amaral Pales Soares. Compilação de semântica denotacional modular. Master's thesis, UFMG, 2007.

[2] M. J. Gordon. *The Denotational Description of Programming Languages - An Introduction*. Springer-Verlag, 1979.

[3] Felipe Silva Loredo and Roberto da Silva Bigonha. Metodologia para adição de plug-in ao ambiente eclipse. Technical report, 2008. Laboratório de Linguagens de Programação, UFMG.

[4] D. Scott. Data type as lattices. *SIAM J. on Comp.*, 1976.

[5] Fabio Tirelo, R. S. Bigonha, and João A. B. Saraiva. Disentangling denotational semantics definitions. *Journal of Universal Computer Science*, 2008.

[6] Fábio Tirelo and Roberto da Silva Bigonha. Notus. Technical report, 2006. Laboratório de Linguagens de Programação, UFMG.