Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciências da Computação

FELIPE SILVA LOREDO

MONOGRAFIA DE PROJETO ORIENTADO EM COMPUTAÇÃO II

Implementação de Semântica Vaga em Notus

Projeto Financiado pela FAPEMIG
Processo CEX-1484/06

Belo Horizonte
2008 / 2º semestre

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

IMPLEMENTAÇÃO DE SEMÂNTICA VAGA EM NOTUS

por
FELIPE SILVA LOREDO

Monografia de Projeto Orientado em Computação II

Apresentado como requisito da disciplina de Projeto Orientado em Computação II do Curso de Bacharelado em Ciência da Computação da UFMG

Prof. Dr. Roberto da Silva Bigonha
Orientador

Belo Horizonte
2008 / 2º semestre

RESUMO

O objetivo do presente trabalho é o desenvolvimento de ferramentas que permitam o uso de definições incrementais na especificação da semântica de linguagens de programação e a apresentação dos benefícios das definições realizadas dessa forma. É possível definir as linguagens de programação de forma completa, coerente, precisa e sem ambigüidades. Utiliza-se a Semântica Denotacional para produzir definições formais, aqui representadas pela linguagem Notus – ferramenta utilizada neste trabalho como base das definições da semântica das linguagens de programação que permite definir os analisadores léxico, sintático e semântico, além da própria semântica das linguagens. Apresentam-se brevemente os recursos da linguagem Notus e o seu compilador, definem-se o conceito e a utilidade das definições incrementais, o conceito de vagueza e de preenchimento da vagueza, e a seguir como esses conceitos serão introduzidos na linguagem Notus para permitir o uso de definições incrementais que controlem a crescente complexidade existente durante a criação de grandes definições.

Palavras-chave: semântica formal, semântica denotacional, linguagem notus, definição incremental, transformadores, vagueza, preenchimento de vagueza.

LISTA DE FIGURAS

LISTA DE SIGLAS

OO	Orientado(a) por Objetos
POO	Programação Orientada por Objetos
TAD	Tipo Abstrato de Dados

SUMÁRIO

1 INTRODUÇÃO

A *Wikipédia* define ciência como uma busca pela verdade através de uma investigação metódica dos fatos. Essa investigação metódica só é possível quando há uma formalização dos passos utilizados para alcançar as conclusões sobre o objeto de estudo, de forma que os resultados obtidos sejam reproduzíveis. Nas ciências exatas, utilizam-se o formalismo e a notação matemática para atingir esse objetivo. A idéia é que durante o estabelecimento de uma conclusão cria-se um embasamento matemático que garanta esses fatos, definindo-os de forma precisa, não ambígua e completa.

No campo das linguagens de programação a Semântica Formal é um método existente para formalizar o significado de programas de computador. Ela se contrapõe aos métodos informais, onde as especificações da linguagem, geralmente em linguagem natural, são propensas a fornecer especificações problemáticas, por serem ambíguas, incompletas, ou incoerentes. Uma importante técnica de especificação é a Semântica Denotacional. Nela a semântica é dada pelo mapeamento das estruturas da linguagem em objetos matemáticos denominados denotações. Baseando-se nessa técnica, [Tirelo e Bigonha] definiram a linguagem Notus. Considerando o ambiente de desenvolvimento dessa linguagem e que a produção de sistemas acontece de forma incremental estabeleceu-se o objetivo deste trabalho. A idéia é que o usuário da linguagem Notus possa produzir uma especificação denotacional minimizando o retrabalho existente devido ao acréscimo de construções possivelmente não cobertas por definições parciais anteriores. Deseja-se que a partir de uma definição inicial seja possível por meio de acréscimos sucessivos a total cobertura de todas as construções da linguagem.

2 REFERENCIAL TEÓRICO

2.1 Semântica Formal

A definição informal da semântica de uma linguagem de programação pode ser útil em vários casos, como na apresentação de linguagens e em práticas didáticas. O problema das definições informais é que elas são propensas a fornecer especificações problemáticas, por serem ambíguas, incompletas, ou incoerentes. A Semântica Formal é o ferramental utilizado para formalizar o sentido dos programas de computador. Trata-se de um conjunto de técnicas para

definição de linguagens de programação que buscam suprimir deficiências existentes nas definições informais.

Considere os exemplos a seguir:

Exemplo 1:

procedure Inc(x: *integer*);

begin

    x := x + 1;

end;

- Definição informal: "*Inc* é um procedimento que incrementa uma variável inteira passada como parâmetro em uma unidade".
- Problema com a definição informal: se considerarmos que a passagem parâmetro é por referência, o que aconteceria com a chamada *Inc*(1)?

Exemplo 2:

x := *f*(2);

- Definição informal: *x* é uma variável inteira e *f* é uma função que recebe um inteiro e retorna também um inteiro. O resultado de *f*(2) é atribuído a variável inteira *x*.
- Problema com a definição informal: o que aconteceria se a função *f* tivesse um comando *goto* e não retornasse?

Os exemplos acima ilustram alguns dos diversos problemas encontrados no uso das definições informais. A Semântica Formal existe para resolver esses problemas, ela utiliza a notação formal e o rigor matemático para tratar os conceitos com maior precisão, sendo utilizada em atividades de importâncias teóricas e práticas, por exemplo, na padronização de linguagens, geração automática de compiladores e projeto de novas linguagens.

2.2 Semântica Denotacional

Dentro da Semântica Formal há uma técnica para especificação conhecida como Semântica Denotacional, nela semântica é dada pelo mapeamento das estruturas da linguagem em objetos matemáticos chamados denotações. A semântica é dada por um conjunto de funções e seus domínios, as construções a serem especificadas são padrões ou casos dessas funções, o resultado dessas construções é a influência da construção no comportamento global do programa, há parâmetros auxiliares de controle que fornecem informações do contexto e as

funções chamam-se recursivamente repassando os parâmetros transformados de forma que ao término do processo o sentido dos programas seja obtido.

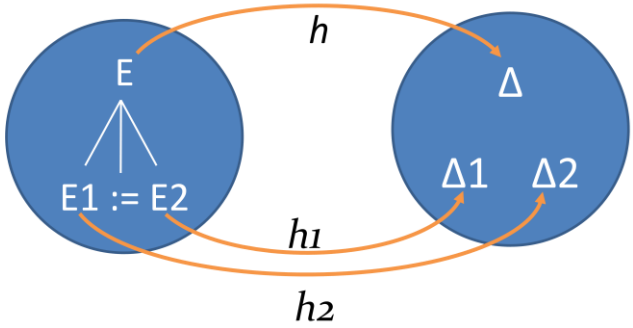

Figura 1 - A semântica do todo é dada pela semântica de seus constituintes

Na Figura 1, se **E** é uma expressão constituída por $\mathbf{E_1}$ **:=** $\mathbf{E_2}$ e **Δ**, $\mathbf{\Delta_1}$, $\mathbf{\Delta_2}$ são respectivamente o significado de **E**, $\mathbf{E_1}$ e $\mathbf{E_2}$, se as funções h_1 e h_2 mapeiam $\mathbf{E_1}$ e $\mathbf{E_2}$ em $\mathbf{\Delta_1}$ e $\mathbf{\Delta_2}$ respectivamente, isto é,

$$h_1(\mathbf{E_1}) = \mathbf{\Delta_1} \text{ e } h_2(\mathbf{E_2}) = \mathbf{\Delta_2}$$

a função *h* de **E** é dada pela aplicação de uma função as denotações de $\mathbf{E_1}$ e $\mathbf{E_2}$, de forma que o significado do todo (**E**) é dada por uma função do significado de seus constituintes ($\mathbf{E_1}$, := e $\mathbf{E_2}$), ou seja,

$$h(\mathbf{E}) = f(h_1(\mathbf{E_1}), h2(\mathbf{E_2}))$$

2.3 Linguagem Notus

Baseando-se na Semântica Denotacional [Tirelo e Bigonha] definiram a linguagem Notus. Trata-se de uma linguagem de domínio específico cujo propósito é prover um ambiente compreensível para a definição modular de linguagens de programação.

Em Notus, é possível definir recursos comuns às linguagens de programação, incluindo a parte léxica, sintática e semântica. Além disso, a linguagem oferece recursos para descrição modular, equivalência estrutural de tipos, controle de visibilidade, encapsulação, ocultação e monâdas.

A divisão dos módulos que compõem a descrição formal de uma linguagem é orientada por sua sintaxe abstrata, onde cada módulo é composto por

- *importações*, onde são enumeradas as dependências do módulo,
- *especificações léxica e sintática*, que definem ou estendem os componentes léxicos e sintáticos de um conjunto de construções,
- *definições de domínios sintáticos e semânticos*, e
- *funções semânticas*.

A especificação léxica de uma linguagem em Notus é realizada por meio de expressões regulares semelhantes às utilizadas por geradores de analisadores léxicos, como o Lex [Lex]. 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 uma ferramenta para a descrição formal de linguagens de programação, Notus possui como principais características:

- 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 mais alta para a construção de equações que especificam a semântica dos constituintes da linguagem;
- e, permitir a escrita de definições com aridades diferentes para uma mesma função.

A seguir apresentamos os principais recursos da linguagem Notus.

Pacotes

Como em *Java*, os módulos de uma definição em Notus podem ser organizados em pacotes, que são conjuntos de módulos. Além disso, pacotes podem conter pacotes que são chamados subpacotes e os pacotes são utilizados como ferramenta constituinte do controle de visibilidade de Notus – *visibilidade pacote*.

Módulo Principal

Toda especificação realizada em Notus possui um módulo principal, denominado *Main*, que define os seguintes elementos:

- função de pré-processamento do arquivo de entrada;

- o símbolo inicial da gramática;
- os arquivos de entrada;
- os arquivos de saída;
- e, a função semântica para avaliação da árvore de sintaxe abstrata.

O módulo principal de uma definição Notus deve possuir o formato:

module Main
... module constituents
syntax s;
... other module constituents
end.

Domínios Sintáticos

Em Notus tipos são domínios de Scott. Esses domínios sintáticos denotam domínios de *tokens* e variáveis de gramática.

Domínios Semânticos

Domínios semânticos em Notus definem o universo de valores usados na especificação da semântica de linguagens de programação. Os domínios semânticos possuem controle de visibilidade e, portanto, podem ser explicitamente declarados como públicos ou privados, e, na ausência de declaração explícita de visibilidade, são visíveis somente pelos módulos de seu pacote.

Os domínios semânticos são definidos por meio de uma expressão composta por operações realizadas sobre domínios básicos. Esses domínios básicos são domínios sintáticos, domínios pré-definidos ou enumeração de domínios. Para a composição de domínios é possível utilizar as operações

- definição de domínio tupla;
- definição de domínio união disjunta;
- definição de domínio lista;
- e, definição de domínio funcional.

Especificação Léxica

Na seção de especificação léxica de um módulo em Notus, definem-se os *tokens* da linguagem descrita. Notus permite a definição de macros para a melhoria da legibilidade e economia de escrita. *Tokens* e macros são definidos por expressões regulares e podem ter controle de visibilidade público ou privado, isto é, podem ser visíveis por todos os módulos da especificação ou somente dentro do módulo em que estão definidos.

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* de *Haskell* que representarão esses 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.

O símbolo inicial da gramática é definido por meio da cláusula *syntax*. O módulo principal da especificação de uma linguagem deve obrigatoriamente possuir a declaração do símbolo inicial, e, além disso, esse símbolo deve ser uma variável de gramática declarada no módulo principal ou uma variável pública declarada em um dos módulos que ele importa.

Especificação Semântica

As especificações semânticas em Notus são compostas pelos domínios semânticos, pelas funções semânticas, e pelas equações que descrevem a semântica dos construtos da linguagem de programação especificada. As equações semânticas são descritas por funções. As definições para uma função determinam o resultado de sua aplicação para uma seqüência de padrões em particular, e podem aparecer em qualquer ordem e em módulos distintos.

Exemplo na Linguagem Notus

A título de ilustração, a seguir apresentamos um exemplo de um programa tipo *Hello World* na linguagem Notus:

```
Module Main
Private syntactic Prog;
token text = .;
prog ::= text;
syntax prog;
output stdout;
semantics mainFunction;
function mainFunction: Prog -> String;
mainFunction p = “Hello Notus World”;
end
```

Hello World na linguagem Notus.

Inicialmente dá-se nome ao módulo, a seguir define-se uma pequena gramática para a linguagem (no caso será utilizada apenas por exigência da linguagem), define-se o símbolo inicial da gramática, define o método de saída, a função que deve ser chamada inicialmente para a definição e por último define-se a única função do exemplo. O compilador Notus gera a partir da definição de uma linguagem três tipos de arquivo: uma especificação *Alex*, outra *Happy* e um conjunto de módulos em linguagem *Haskell*. A seguir, a título de ilustração, apresentamos essas saídas para o exemplo acima.

Especificação léxica: Lexer.x

```
{
module Lexer where
import NotusDefault
import DataModule
import NotusFunctions

}

%wrapper "basic"

tokens :-

.                   { \s->(T__main__text(Text__0 ( s)))}

{
-- The token type:

data Token__Main = T__main__text Text__Main
          deriving Eq

data Tree a b = Leaf (a,b)
       | Node (a,b) (Tree a b) (Tree a b)
       | Null
   deriving Eq
```

```
lookupT :: Ord a => Tree a Token__Main -> a -> Token__Main -> Token__Main
lookupT (Leaf (a,b)) s tReturn
  | s == a = b
  | otherwise = tReturn
lookupT (Node (n,t) l r) s tReturn
  | n == s  = t
  | s > n   = lookupT r s tReturn
  | s < n   = lookupT l s tReturn
lookupT Null _ tReturn = tReturn
```

Especificação Sintática – Syntactic.y

```
{
module Syntactic where
import Char
import Lexer
import NotusDefault
import DataModule
import NotusFunctions
}

%name parse main__prog
%tokentype { Token__Main }

%token
        main__text                      { T__main__text $$ }
%%
main__prog    : main__text              { Prog__1 $1 }

{

happyError :: [Token__Main] -> a
happyError _ = error "Parse error"


--------------------Ignore Indirections Functions--------------------

}
```

Módulo principal – Main.hs

```
module Main where

import Lexer
import Syntactic
import DataModule
import NotusDefault
import NotusFunctions
import Data.Bits
import System

--------- Functions ---------

main__mainFunction :: (Prog__Main -> String__NotusDefault)
main__mainFunction = ( (\prog__main__3-> "Hello Notus World" ))




main = do
  (x:xs) <- getArgs
  progL <- readFile x
  sWriterNotus (head(matchOutFiles [(Stdout)] xs))(main__mainFunction (parse
(alexScanTokens progL)))
```

2.4 Definições Incrementais – Transformadores

A definição de todas as construções de uma linguagem de uma só vez é algo pouco prático, difícil de ser feito e ainda mais difícil de ser compreendida. Dessa forma, é desejável que as definições possam ser feitas de maneira incremental. Trata-se do conceito de vagueza: algumas construções são deixadas em aberto ou não definidas, para que posteriormente sua semântica seja dada. Deve-se observar como essa abstração é amplamente utilizada nos diversos campos da ciência. Ao construir uma casa as paredes são colocadas de pé sem as mangueiras de energia. Sabe-se que posteriormente cabos de energia, telefone e outros serão instalados, mas momentaneamente eles são, apesar de sua existência ser conhecida, ignorados. Ocasionalmente há alguma preparação para que as mangueiras sejam adicionadas posteriormente, mas a regra geral é que as paredes serão quebradas para a adição das mangueiras. De maneira análoga, a definição de uma linguagem deveria poder abstrair partes que são momentaneamente irrelevantes, para que posteriormente sejam consideradas. Durante a extensão da definição incremental de uma linguagem, algumas partes de definições

anteriores podem precisar ser redefinidas por possíveis necessidades de novas construções. A extensibilidade de Notus trata justamente deste problema. Na linguagem é possível realizar essas extensões evitando boa parte do retrabalho através do uso de um recurso conhecido como transformação de módulos. Produz-se uma especificação incremental com controle da complexidade da definição, de forma que a especificação da linguagem possa evoluir de algo mais simples até a total cobertura de todas as suas estruturas. Este trabalho tratará da adição de novas construções ao compilador da linguagem Notus para permitir a transformação de módulos.

Dada uma definição original D_0, deseja-se obter uma definição transformada D_1 onde

$D_1 = T(D_0) + \Delta$,

T é a função de transformação que recebe a definição original e fornece como resultado uma definição D_0' em que as funções de D_0 foram alteradas conforme um conjunto de regras especificadas em T e Δ é o conjunto de domínios, funções, regras de gramática e tokens adicionados para compor D_1.

2.5 Transformação de Módulos

A transformação de módulos é o recurso existente na linguagem Notus para permitir as definições incrementais. É um recurso para quebrar a complexidade em partes mais bem compreensíveis e gerenciáveis, contrastando com um tratamento centrado no todo. A partir de uma definição básica agregam-se mais e mais construções até que toda a linguagem alvo esteja coberta. Veja a equação dos transformadores:

$$D_1 = Transf(D_0) + \Delta$$

Dada uma definição original D_0, a definição D_1 que estende D_0, é a definição original D_0 transformada pela função de transformação *Transf* (isto é, $Transf(D_0)$) mais um conjunto de definições adicionais Δ.

Em Notus é um conjunto de quatro cláusulas que possibilitam alterar o comportamento das funções de definições anteriores, permitindo que novos constituintes das definições possam ser adicionados sem que seja necessário re-escrever todas as funções envolvidas com a nova construção. A seguir apresentamos a sintaxe dos transformadores existentes:

transformation *transformation-name*

signature	$f_1 : T_1$ **to** $T'_1, f_2 : T_2$ **to** $T'_2, \cdots, f_m : T_m$ **to** T'_m
default	$l_1 = c_1, l_2 = c_2, \cdots, l_n = c_n$
replace	$f_1\ p_{11}\ \cdots\ p_{1k_1}$ **by** $e_1, \cdots, f_n\ p_{n1}\ \cdots\ p_{nk_n}$ **by** e_n
redefine	$f_1\ p_{11}\ \cdots\ p_{1k_1}\ =\ e_1, \cdots, f_n\ p_{n1}\ \cdots\ p_{nk_n}\ =\ e_n$

Signature

Permite alterar a assinatura de uma função incluindo novos parâmetros. Funções com chamadas recursivas têm o valor de seus parâmetros propagados de forma inalterada para cada chamada.

Default

Permite definir um valor padrão para os novos parâmetros adicionados pela cláusula ***signature***. É o recurso utilizado para atribuir valores aos parâmetros que não são propagados dentro das funções.

Replace

Decoração de função. Dado um padrão, toda vez que ele ocorrer numa chamada de uma função a expressão original da função é substituída por uma nova expressão.

Redefine

Permite redefinir uma função. Sua vantagem é que os arquivos da definição original permanecem inalterados, mas na definição atual a função é re-escrita conforme os especificado por esta cláusula. Necessário para casos extremos em que os recursos anteriores não são suficientes.

2.6 Exemplos de Transformadores

A seguir apresentamos em caráter ilustrativo dois exemplos do uso de transformadores.

Exemplo 1

Considere:

- a definição original D_0:

```
module Main
private syntactic Prog;
token text = .;
prog ::= text;
syntax prog;
output stdout;
semantics mainFunction;
function mainFunction: Prog -> String;
mainFunction p = (showMessage "hello") ++ (showMessage "notus");
function showMessage: String -> String;
showMessage s = s;
end
```

Exemplo de Transformação – Definição inicial D_0

- módulo principal da definição extensora D_1:

```
extension Transf(Hello)
end
```

- módulo que define a transformação *transf*:

```
transformation Transf
    replace showMessage "hello"
            by showMessage "ola!";
end
```

Deve-se observar que esse exemplo não possui a parte Δ da equação dos transformadores. Caso ela fosse necessária bastaria adicionar um *import* ao módulo principal da definição extensora.

Exemplo 2:

Suponha que após ter escrito as equações para todos os comandos de uma linguagem decida-se incluir a ela o comando *break*. Na abordagem tradicional <u>todas</u> as equações relativas a comandos precisariam ser descartadas e re-escritas. A seguir apresentam-se as alterações que seriam necessárias para a inclusão do *break* usando-se transformação (somente a parte relevante a discussão é apresentada).

Abstract syntax:

$$P \in Prog \rightarrow D; C$$
$$C \in Com \rightarrow \textbf{while}\ E\ C \mid C_1; C_2 \mid \cdots$$
$$E \in Exp \rightarrow \cdots$$
$$D \in Dec \rightarrow \cdots$$

Semantic Functions:

$$\mathcal{P} : Prog \rightarrow Ans$$
$$\mathcal{C} : Com \rightarrow Env \rightarrow Cc \rightarrow Cc$$
$$\mathcal{E} : Exp \rightarrow Env \rightarrow (Val \rightarrow Cc) \rightarrow Cc$$
$$\mathcal{D} : Dec \rightarrow Env \rightarrow (Env \rightarrow Cc) \rightarrow Cc$$

Semantic Equations:

$$\mathcal{P}[\![D; C]\!] = \mathcal{D}[\![D]\!]\ r_0\ (\lambda r.\mathcal{C}[\![C]\!]\ r\ (\lambda s.stop))\ s_0$$
$$\mathcal{C}[\![C_1; C_2]\!]\ r\ c = \mathcal{C}[\![C_1]\!]\ r;\ \mathcal{C}[\![C_2]\!]\ r\ c$$
$$\mathcal{C}[\![\textbf{while}\ E\ C]\!]\ r = \text{FIX}\ \lambda f c.\mathcal{E}[\![E]\!]\ r;\ \lambda v.\textbf{if}\ v\ \textbf{then}\ \mathcal{C}[\![C]\!]\ r\ (f\ c)\ \textbf{else}\ c$$

Other equations defining $\mathcal{C}$, $\mathcal{E}$, and $\mathcal{D}$

Assim uma possibilidade para adição do *break* seria escrever o transformador

$$\textbf{transformation}\ include\_break\_a$$
$$\textbf{signature}\ \ \mathcal{C} : Com \rightarrow Env \rightarrow b : Cc \rightarrow Cc \rightarrow Cc$$
$$\textbf{default}\ \ b = \lambda s.error$$

E adicionar a equação do próprio *break*

$$\mathcal{C}\ [\![\textbf{break}]\!]\ r\ b\ c = b$$

A título de ilustração, a seguir apresentam-se as equações alteradas pela aplicação da transformação acima:

$$\mathcal{C} : Com \rightarrow Env \rightarrow Cc \rightarrow Cc \rightarrow Cc$$
$$\mathcal{P}[\![D; C]\!] = \mathcal{D}[\![D]\!]\ r_0\ (\lambda r.\mathcal{C}[\![C]\!]\ r\ (\lambda s.error)\ (\lambda s.stop))\ s_0$$
$$\mathcal{C}[\![C_1; C_2]\!]\ r\ b\ c = \mathcal{C}[\![C_1]\!]\ r\ b;\ \mathcal{C}[\![C_2]\!]\ r\ b\ c$$
$$\mathcal{C}[\![\textbf{while}\ E\ C]\!]\ r\ b\ c = (\text{FIX}\ \lambda f c'.\mathcal{E}[\![E]\!]\ r;\ \lambda v.\textbf{if}\ v\ \textbf{then}\ \mathcal{C}[\![C]\!]\ r\ b\ (f\ c')\ \textbf{else}\ c')\ c$$

3 METODOLOGIA

O primeiro passo para a realização do projeto foi a compreensão do compilador Notus que possui código extenso e uso de programação orientada a aspectos. A etapa seguinte foi definir a forma como o compilador deveria ser alterado:

- Alterar analisador léxico;
- Alterar analisador sintático;

- Adicionar a semântica das transformações: mudar funções de definições anteriores.

Era necessário não apenas acrescentar as construções aos analisadores, mas sim adicioná-las de forma que elas fossem compatíveis com àquelas que já haviam sido criadas anteriormente.

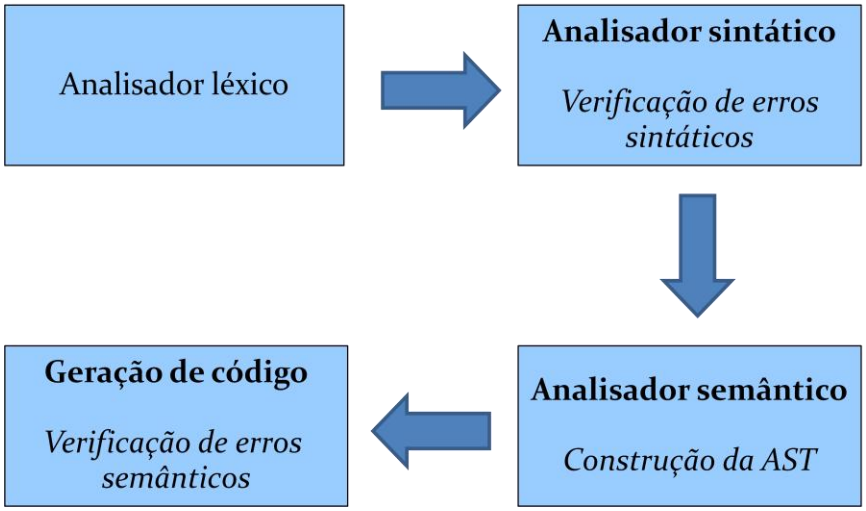

Figura 2 - Pontos de alteração do compilador Notus

A Figura 2 representa os pontos em que o compilador Notus foi alterado. Para o analisador léxico, sintático e semântico foi necessário adicionar as construções para transformação mantendo-as compatíveis com as estruturas anteriores. Para o analisador sintático, o semântico e o gerador de código foi necessário criar as estruturas para armazenar os transformadores, *instanciar* essas estruturas e preenchê-las, e alterar AST de forma que as informações dos transformadores fossem agregadas a ela. Além disso, foi necessário incluir à verificação de erros semânticos código relativo aos transformadores.

As transformações foram codificadas de forma que houvesse um mapeamento delas em construções da própria linguagem. Por exemplo, toda vez que uma transformação do tipo *replace* era encontrada ela era mapeada para um conjunto de comandos *case*. Trata-se de uma alteração na árvore de sintaxe abstrata. A árvore original do programa é *transformada* de maneira que ela reflita as transformações a que o código foi submetido. O objetivo disso é simplificar a geração de código dos transformadores, pois dessa forma o próprio compilador resolveria questões relativas a geração de código.

3.1 Detalhes de projeto

A seguir apresentamos com mais detalhes o que foi feito no desenvolvimento do projeto.

Analisador léxico

No desenvolvimento do compilador Notus utilizou-se JFlex e uma lista encadeada com todos os tokens a serem reconhecidos pelo analisador. Dessa forma, a alterações necessárias limitaram-se a adicionar os tokens *transformation, signature, default, replace, redefine, by, extension* e *contains.* Como era de se esperar, na relação de ferramentas tipo Lex e Yacc, foi necessário declarar tais tokens no analisador sintático.

Analisador sintático

O compilador Notus utiliza um esquema de análise sintática em que há um analisador sintático que reconhece os elementos sintáticos do código fonte do usuário e uma entidade auxiliar que é responsável por criar e armazenar em memória os elementos reconhecidos, veja a figura 3:

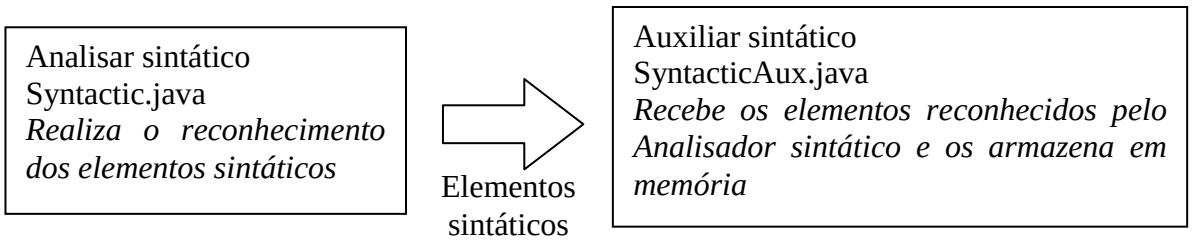

Figura 3 – Esquema de funcionamento do analisador sintático

Assim, as alterações necessárias na parte sintática do compilador foram adicionar as regras
transformationModule ::= transfHeader importsSection transfSectionList END;

transfSectionList ::= transfSectionList:list transfSection:sec
| ;

transfHeader ::= TRANSFORMATION domainIdentifier:id;

transfSection ::= SIGNATURE signatureClauseList:sig SEMI
| DEFAULT defaultClauseList:def SEMI

| REPLACE replaceClauseList:rep SEMI
| REDEFINE redefineClauseList:red SEMI;

signatureClauseList ::= signatureClause: sig
| signatureClauseList COMMA signatureClause;

signatureClause ::= ID:f COLON labelledTypeList: list;

labelledTypeList ::= lTConstituent:lTConst
| labelledTypeList:list ARROW lTConstituent:lTConst;

lTConstituent ::= instDomainExp:l
| ID:label COLON instDomainExp:l;

defaultClauseList ::= defaultClause:def
| defaultClauseList COMMA defaultClause;

defaultClause ::= ID:label EQUAL expression:exp;

replaceClauseList ::= replaceClause:rep;
| replaceClauseList COMMA replaceClause;

replaceClause ::= ID:f functionParams:p BY expression:exp;

redefineClauseList ::= ID:f functionParams:p BY expression:exp;

E a regra *module* já existente no analisador sintático precisou ser alterada para que o suporte aos módulos de transformação fosse adicionado:

module ::= moduleHeader:m importsSection:imports declaration:decs moduleEnd;
// Transformation
| transformationModule
| extensionHeader containsSection moduleEnd;

A classe *SyntacticAux* foram adicionados os métodos:

- *public static TLabelledType createLabelledType(DomainExpression dExp, Identifier label)* – responsável pela criação do tipo que possui um label da cláusula *signature* utilizado para trabalhar em conjunto com a cláusula *default*.
- *public static TDefault createDefaultClause(Identifier label, Expression exp)* – responsável pela criação da cláusula *default*. Armazena o label do parâmetro que a cláusula se refere e a expressão constante que ela representa.
- *public static Transformer createTransformer(Identifier id, Identifier packageId, Identifier previousSpecification)* – cria uma ocorrência da classe que armazena todo o conjunto de cláusulas de transformação de uma definição.
- *public static TReplace createReplaceClause(Identifier funName, ArrayList <FunctionPattern> patternList, Expression expression)* – cria uma ocorrência da classe que armazena os constituintes da cláusula *replace*. Possui o identificador da função e o padrão a que cláusula se aplica, além da expressão que deve substituir a expressão já existente.
- *public static TRedefine createRedefineClause(Identifier funName, ArrayList <FunctionPattern> patternList, Expression expression)* – cria uma ocorrência da classe que armazena os constituintes da cláusula *redefine*. Contém o identificador da função e o padrão a que a cláusula se aplica, além da expressão que deve substituir a expressão já existente.

Analisador semântico

O analisador semântico do compilador possui as seguintes etapas:

1. Com as informações sintáticas recolhidas, inicialmente o conjunto de funções *built-in* da linguagem são criadas e armazenadas na memória. Elas não possuem distinção das funções definidas pelo usuário, exceto é claro, pelo fato de que são definidas pelo próprio compilador.
2. Baseando-se nas definições léxicas da especificação do usuário o analisador gera a saída no formato do *Alex* que posteriormente deverá ser convertida em código *Haskell*, isto é, gera-se a parte léxica do compilador do usuário.
3. A partir da gramática informada pelo usuário o analisador semântico gera a saída no formato do *Happy* que posteriormente será convertida em código *Haskell*, ou seja, a parte sintática do compilador do usuário é gerada.

4. A partir das definições de funções do usuário a parte semântica do compilador do usuário é gerada. As funções em linguagem Notus são mapeadas para suas correspondentes em *Haskell* respeitando-se a organização em pacotes do código fonte do usuário. As funções são compiladas para *Haskell* de forma que seu nome seja único, adicionando ao nome simples da função toda seqüência de pacotes que compõem seu nome completo, por exemplo, uma função *f* contida no pacote *A.B.C.D* seria mapeada para algo como *A__B__C__D__f*.

Os transformadores foram programados para serem aplicados as definições antes da segunda etapa, pois a partir dela as funções poderiam ser necessárias para gerar as saídas em Haskell. Criou-se uma função *runPreviousSpecification* que compila definições anteriores para que depois a possível transformação da especificação atual seja aplicada. Essa função é chamada recursivamente até que todas as definições anteriores estejam compiladas e a atual possa ser processada. Dentro dessa função chama-se *runTransformationModule* que é a responsável pelo processamento do módulo de transformação, ela chama o analisador léxico e sintático para o módulo de transformação e como resultado disso, preenche as estruturas de dados relativas a transformação. Na seqüência em *runPreviousSpecification* o método *applyTransformations* a invocado para que as transformações lidas do módulo de transformação sejam aplicadas a definição anterior. A seguir apresentamos em alto nível o algoritmo de *applyTransformations*:

applyTransformations
    Para toda declaração de função
        Alterar a declaração de acordo com as cláusulas signature
        Alterar os padrões de todas as definições da função

        Para toda definição da função
            Alterar a expressão da definição de acordo com signature
            Alterar a expressão da definição de acordo com replace
        fim para

        Para cada cláusula redefine
            Alterar a declaração da função de acordo com o redefine
        fim para
    fim para

fim applyTransformations

Tipos criados para os transformadores

O desenvolvimento foi iniciado com a programação de duas cláusulas de transformação: *signature* e *default*. Isso foi feito porque ambas trabalham em conjunto e serviriam de base para o desenvolvimento das restantes. Ambas foram codificadas de forma que elas alterassem a árvore de sintaxe abstrata e a deixassem em um estado válido para que o compilador pudesse utilizá-la para geração de código.

TDefault – representa a cláusula *default* dos transformadores. Foi criada para agrupar e armazenar o *label* e a *expressão* da cláusula *default*.

TSignature – contém a função a ser transformada, o número de parâmetros que serão adicionados pela cláusula e uma lista com a assinatura da função após a transformação. A lista é composta por elementos de dois tipos: *types* e *labelled types*. O primeiro indica parâmetros inalterados, isto é, os parâmetros originais da função e são simplesmente tipos como *Int*, *String*, etc. *Labelled types* representam tipos rotulados que foram adicionados a definição da função, ou seja, tratam-se dos novos parâmetros da função. *TSignature* possui três métodos importantes:

- *changeFunctionDefinition* – percorre os padrões da função a ser transformada adicionando os novos parâmetros.
- *changeFunctionDeclaration* – altera a declaração (cabeçalho) da função a ser transformada adicionando os novos parâmetros.
- *createLambdaAbstraction* – substitui a expressão da definição da função a ser transformada por uma expressão lambda equivalente que leva em consideração a transformação a ser realizada:
    - Se a expressão é uma chamada recursiva da função a ser transformada, os parâmetros adicionados são propagados para a chamada recursiva.
    - Caso contrário o parâmetro é substituído na expressão por seu equivalente especificado na cláusula *default*.

Terminada a programação das cláusulas *default* e *signature* passou-se para o desenvolvimento da cláusula *replace*. Para ela era necessário percorrer a expressão de todas as definições de função procurando as chamadas da função referenciada por *replace* e nesses casos verificar se os parâmetros da chamada casavam com os padrões definidos na cláusula de transformação. Devido à complexidade da tarefa e por poder depender de parâmetros disponíveis apenas durante a execução, a transformação foi codificada substituindo-se a chamada da função por

uma expressão lambda equivalente em que os parâmetros da chamada eram submetidos por um comando *case* de Notus que faz o casamento de padrões.

TReplace – classe responsável pelo armazenamento da expressão, da função e do padrão a que a cláusula *replace* se aplica. Além disso, seu principal método é responsável, recebendo a declaração de uma função, pela criação da expressão *case* que substituirá a expressão original da definição da função.

A última cláusula codificada foi a *replace*. A mais complicada, nela foi necessário criar uma definição de função e armazená-la para posteriormente substituir a definição cujo padrão casasse com a especificada em *replace*. Além disso, era necessário casar o padrão de cada definição de forma que os compatíveis resultassem na substituição da definição. Como o casamento de padrões é uma tarefa complexa e que demanda informações disponíveis somente após o início da geração de código, parte dessa geração precisou ser adiantada para que fosse possível utilizar o sistema de verificação de padrões já existente, exatamente o que é realizado no principal método do tipo.

TRedefine – classe responsável pelo armazenamento da definição de função que substituirá a definição cujo padrão casar com o especificado na cláusula *redefine*.

Além desses tipos criou-se também uma classe chamada *Transformer* que armazena todas as cláusulas de transformação a serem aplicadas a definição, o identificador da transformação, o pacote onde ela se encontra e o identificador da especificação anterior.

4 RESULTADOS E DISCUSSÃO

Ao término do projeto obteve-se uma versão funcional do compilador com suporte a transformação. A execução da tarefa mostrou-se bastante complexa, dados o tamanho do código do compilador, a quase total ausência de comentários e a utilização de programação orientada a aspectos. Houve muita dificuldade em compreender como o compilador funcionava. Em alguns casos a depuração era impraticável, pois as ferramentas não estavam preparadas para lidar com a existência dos aspectos – não havia execução passo a passo, ou ela não correspondia ao código escrito. Outro grande foco de dificuldade foi a verificação de erros semânticos, pois ela é complexa e realizada de forma tardia no sistema. Foi necessário muito tempo para que as construções para transformação fossem tratadas adequadamente por essa parte do compilador.

Como prova conceito realizou-se um conjunto de experimentos sob o sistema obtido. Realizou-se uma definição incremental de parte da linguagem Java (ela não foi totalmente

coberta devida sua grande extensão) e criaram-se cerca de 10 testes simples para verificar a correção dos recursos desenvolvidos. Em alguns casos problemas foram encontrados e sua solução foi providenciada, após essas revisões todos os testes foram realizados com sucesso.
Outro fator complicador foi a existência de erros no compilador que dificultavam o avanço do trabalho que por várias vezes precisou ter seu foco alterado da inclusão dos transformadores a ferramenta para correção de erros da própria ferramenta. Além disso, os transformadores foram codificados de forma que as alterações que eles realizam sobre o código do usuário fossem mapeadas em construções da própria linguagem, algo que era uma vantagem por simplificar a tarefa a ser executada por várias vezes foi um complicador pela existência de erros nos recursos utilizados.

5 CONCLUSÕES E TRABALHOS FUTUROS

A parte mais complexa do trabalho executado foi a compreensão do compilador Notus dada a alta complexidade da ferramenta, os recursos utilizados no seu desenvolvimento e o acoplamento existente. A programação do recurso em si não consumiu muito tempo e não foi muito complexa, excetuando-se os casos em que alguns recursos necessários da linguagem para produzir os transformadores apresentavam erros de programação, casos em que sua solução teve que ser providenciada ou outro recurso foi utilizado para contornar o problema.
A inclusão dos transformadores ao compilador Notus foi realizada com sucesso apesar das dificuldades encontradas. Especificações testes foram realizadas como prova de conceito. Durante o desenvolvimento dos transformadores e a realização desses testes produziu-se uma lista de erros existentes no compilador que fornecem um passo inicial para trabalhos futuros.

6 APÊNDICE I – CÓDIGO FONTE

A seguir são apresentados métodos e classes criadas e alteradas para o desenvolvimento do projeto.

6.1 Classe Notus

```
// Transformation
private void runPreviousSpecification(String moduleFileName, String moduleName) throws
Exception
{
   String specName;
   // Compila módulo principal (da especificacao corrente)
   runMainModule(moduleFileName, moduleName);

   // Trata das transformacoes
   // Deve repetir a compilacao do principal ate encontrar
   // o arquivo Main que nao contenha um extension
   if (specification.getMainModule() == null)
   {
      // Nome da especificacao anterior (sendo transformada)
      specName =
specification.getCurrentTransformation().getPreviousSpecification().getName();
      // Nome do modulo main da especificao anterior para
      // possivel emissao de mensagem de erro
      moduleName = specName + java.io.File.separator + Notus.mainModuleName;


      // Volta dois diretorios: ESPECIFICACAO_ATUAL\SRC e
      // a seguir entra no da especificacao anterior (estendida)
      File newMain = new File(specification.getRootPackage().getId().getName() +
            java.io.File.separator + ".." +
            java.io.File.separator + ".." +
            java.io.File.separator + specName +
            java.io.File.separator + "src" +
            java.io.File.separator + Notus.mainModuleName + ".nts");
      moduleFileName = newMain.getCanonicalPath();

      // A especificacao anterior existe?
      if (! newMain.exists())

notus.Error.printError(TransformerMessages.PREVIOUS_SPECIFICATION_NOT_FOUND
_B +
            moduleFileName +
            TransformerMessages.PREVIOUS_SPECIFICATION_NOT_FOUND_M +
            specName +
            TransformerMessages.PREVIOUS_SPECIFICATION_NOT_FOUND_E);

      runPreviousSpecification(moduleFileName, moduleName);

      // Compila modulo de transformacao
```

```
        runTransformationModule(specification.getCurrentTransformation());

        // Transforma as funcoes de acordo com as transformacoes
        applyTransformations();
    }

    //Compila modulos pendentes
    int nModsToCompiles = specification.getUncompiledModules().size();
    while(nModsToCompiles > 0){
        runUncompiledModules();
        nModsToCompiles = specification.getUncompiledModules().size();
    }

    //Tenta resolver os identificadores pendentes em Specificatio.pendencies
    resolveUndefinedIds();

    //Resolve extensoes de variaveis pendentes
    resolveVarsExtensions();

    // Resolve extensões de tokens pendentes
    resolveTokenExtensions();

    //Resolver extensoes de macros pendentes
    resolveElementExtensions();

    //Resolver extrensoes de dominio semantico pendentes
    resolveSemDomExtensios();

    //Resolver definições de funções pendentes
    resolveFunctionDefinitions();

    //Resolver expressões de contexto pendentes
    resolveContextExtensions();

    //Verificar visibilidade de usos e definições
    verifyVisibility();
    }

    private void applyTransformations()
    {
    List<FunctionDeclaration> funDecs =
SemGenerator.getInstance().getFunctionDeclarationList();
    Transformer transf = specification.getCurrentTransformation();
    ArrayList<TSignature> allSigs = transf.getSignatures();
    ArrayList<TRedefine> allReds = transf.getRedefines();

    // Varre as declaracoes das funcoes
    for (FunctionDeclaration funDec : funDecs)
    {
```

```
        // Muda os cabecalhos e padroes das funcoes
        for (TSignature s: allSigs)
        {
        Identifier sigId = s.getFunctionId();
        String fullSigName = ((FunctionDeclaration)sigId.getRole()).getFullName();

        // Nomes completos iguais?
        if (fullSigName.compareTo(funDec.getFullName()) == 0)
        {
           // Sim, mudar o cabecalho
           s.changeFunctionDeclaration(funDec);

           // Mudar os padroes
           for (FunctionDefinition funDef : funDec.getFunDefs())
           {
           s.changeFunctionDefinition(funDef);
           }
        }
        }

        // Varre as definicoes de cada funcao
        for (FunctionDefinition funDef : funDec.getFunDefs())
        {
        Operation exp = (Operation)funDef.getExpression();
        funDef.setExpression((Expression)exp.transformForSignature(funDec));
        funDef.setExpression((Expression)exp.transformForReplace(funDec));
        } // end for (FunctionDefinition funDef : funDec.getFunDefs())

        /* ATENCAO: FOI ENVIADO PARA DENTRO DE SEMGENERATOR
         * PORQUE ERA NECESSARIO QUE OS PADROES JA TIVESSEM SIDO
         * RESOLVIDOS - DESTINO: SemGenerator.genFunctionBody */
        // Aplica os redefines
        for (TRedefine r: allReds)
        {
        r.applyRedefine(funDec);
        }
    }
}

public void runSpecification() throws Exception{
        createNotusDefaultModule();
        specification.setNotusDefaultModule(notusDefaultModule);

        //createBuiltInIdentifiers();
        specification.setBuiltInIdentifier(builtInIdentifiers);
        specification.setBinOperatorCodeGenMAP(binOperatorCodeGenMAP);
        specification.setUnaryOperatorCodeGenMAP(unaryOperatorCodeGenMAP);
        specification.setBinOpTypeMap(binOpTypeMap);
        specification.setUnaryOpTypeMap(unaryOpTypeMap);
```

```
        BuiltInSyntactic builtInSyntactic = new BuiltInSyntactic();
        //BuiltInSyntactic.runBuiltInFile("../lib/BuiltIn");
        builtInSyntactic.runBuiltInFile("BuiltIn");

        // Transformation
        runPreviousSpecification(mainModuleFile, Notus.mainModuleName);
    }
```

6.2 Arquivo Lexer

```
/*      Felipe Silva Loredo - 2007-10-18
        %unicode added
*/

package lexical;

import java_cup.runtime.Symbol;
import java.util.HashMap;
import syntactic.*;
import syntactic.ast.*;
import notus.symbolTable.*;
import notus.Module;
import notus.Specification;
import java.util.ArrayList;

%%
%{
        private HashMap<String,Integer> idChildStrings = new HashMap<String,Integer>();

        public int getLine()
        {
                return yyline;
        }

        private void addIdChildren() {
                idChildStrings.put("module",Tokens.MODULE);
                idChildStrings.put("end",Tokens.END);
                idChildStrings.put("import",Tokens.IMPORT);
                idChildStrings.put("token",Tokens.TOKEN);
                idChildStrings.put("public",Tokens.PUBLIC);
                idChildStrings.put("private",Tokens.PRIVATE);
                idChildStrings.put("element",Tokens.ELEMENT);
                idChildStrings.put("is",Tokens.IS);
                idChildStrings.put("ignore",Tokens.IGNORE);
                idChildStrings.put("extend",Tokens.EXTEND);
                idChildStrings.put("syntax",Tokens.STARTSYMBOL);
                idChildStrings.put("with",Tokens.WITH);
                idChildStrings.put("syntactic",Tokens.SYNTACTIC);
                idChildStrings.put("function",Tokens.FUNCTION);
                idChildStrings.put("true",Tokens.TRUE);
                idChildStrings.put("false",Tokens.FALSE);
```

```
                idChildStrings.put("if",Tokens.IF);
                idChildStrings.put("then",Tokens.THEN);
                idChildStrings.put("else",Tokens.ELSE);
                idChildStrings.put("case",Tokens.CASE);
                idChildStrings.put("of",Tokens.OF);
                idChildStrings.put("let",Tokens.LET);
                idChildStrings.put("in",Tokens.IN);
                idChildStrings.put("where",Tokens.WHERE);
                idChildStrings.put("or",Tokens.OR);
                idChildStrings.put("and",Tokens.AND);
                idChildStrings.put("mod",Tokens.MOD);
                idChildStrings.put("not",Tokens.NOT);
                idChildStrings.put("empty",Tokens.EMPTY);
                idChildStrings.put("evaluate",Tokens.EVALUATE);
                idChildStrings.put("semantics",Tokens.EVALUATION);
                idChildStrings.put("preproc",Tokens.PREPROC);
                idChildStrings.put("input",Tokens.INPUT);
                idChildStrings.put("output",Tokens.OUTPUT);
                idChildStrings.put("append",Tokens.APPEND);
                idChildStrings.put("stdin",Tokens.STDIN);
                idChildStrings.put("stdout",Tokens.STDOUT);
                idChildStrings.put("context",Tokens.CONTEXT);
                idChildStrings.put("persistent",Tokens.PERSISTENT);
                idChildStrings.put("ephemeral",Tokens.EPHEMERAL);
                idChildStrings.put("match",Tokens.MATCH);
                idChildStrings.put("at",Tokens.AT);
                idChildStrings.put("from",Tokens.FROM);

                // Adicionados para as transformacoes
                idChildStrings.put("transformation",Tokens.TRANSFORMATION);
                idChildStrings.put("signature",Tokens.SIGNATURE);
                idChildStrings.put("default",Tokens.DEFAULT);
                idChildStrings.put("replace",Tokens.REPLACE);
                idChildStrings.put("redefine",Tokens.REDEFINE);
                idChildStrings.put("by",Tokens.BY);
                idChildStrings.put("extension",Tokens.EXTENSION);
                idChildStrings.put("contains",Tokens.CONTAINS);

        }

        private Symbol checkChildren (String lexeme) {
                Integer token = idChildStrings.get(lexeme);
                if (token != null)
                        return (new Symbol(token));
                else {
                        //Identifier id = SymbolTableOLD.getInstance().lookup(lexeme);
                        Module currentModule =
Specification.getInstance().getCurrentModule();
                        Identifier id = null;
                        if(currentModule != null){
```

```
                                id = currentModule.lookupModule(lexeme,new
ArrayList<Module>());
                        }
                        if (id == null){
                                id = new Identifier(lexeme, new
UndefinedIdentifier(),currentModule);
                                if(currentModule != null)
                                        currentModule.addNewId(lexeme,id);
                                //SymbolTableOLD.getInstance().insert(lexeme,id);
                        }
                        return new Symbol(Tokens.ID,id);
                }
        }


%}
%init{
                addIdChildren();
%init}
%cup
%class Lexer
%public
%state COMMENTS
%state COMMENT
%line
%unicode

charsymbol = [^\n\t\r\f\b\\\'\"]
escapechar = \\[abrfntv\"\\\']
csEscapechar = \\[abrfntv"^""[""]"\"\\\'-]
cscharsymbol = [^\n\t\r\f\b"^""[""]"\\\'\"-]
csInterval = (({cscharsymbol}|{csEscapechar})"-"({cscharsymbol}|{csEscapechar}))
charset = "$[""^"?({csEscapechar}|{cscharsymbol}|{csInterval})+"]"
d = [0-9]
ds = {d}+
l = [a-fA-F]
letterL = [lL]
letterE = "e" | "E"
hexaOrOctal = "0"[xX]({d}|{l})+
intNum = ({ds}) | ({hexaOrOctal})
longNum = {intNum} {letterL}
doubleNun = {d}+("."{d}+)?({letterE}[+-]?{d}+)?
id = [a-z_][0-9a-zA-Z_]*
quoted = [\"]({charsymbol}|{escapechar})*[\"]
plusOrTimes = ("+"|"*")
charsetFUNCIONA = "$[""^"?(.)+"]"

%%
<YYINITIAL> "&"   { return new Symbol(Tokens.BWAND); }
<YYINITIAL> "<"   { return new Symbol(Tokens.LT); }
```

```
<YYINITIAL> "<<"  { return new Symbol(Tokens.SHIFT_L); }
<YYINITIAL> "<="  { return new Symbol(Tokens.LTE); }
<YYINITIAL> ">"   { return new Symbol(Tokens.GT); }
<YYINITIAL> ">>"  { return new Symbol(Tokens.SHIFT_R1); }
<YYINITIAL> ">>>" { return new Symbol(Tokens.SHIFT_R2); }
<YYINITIAL> ">="  { return new Symbol(Tokens.GTE); }
<YYINITIAL> "->"  { return new Symbol(Tokens.ARROW); }
<YYINITIAL> "<-"  { return new Symbol(Tokens.LEFTARROW); }
<YYINITIAL> ";"   { return new Symbol(Tokens.SEMI); }
<YYINITIAL> "~"   { return new Symbol(Tokens.TILDE); }
<YYINITIAL> ","   { return new Symbol(Tokens.COMMA); }
<YYINITIAL> ":"   { return new Symbol(Tokens.COLON); }
<YYINITIAL> "="   { return new Symbol(Tokens.EQUAL); }
<YYINITIAL> "^"   { return new Symbol(Tokens.COMPLEMENT); }
<YYINITIAL> "*"   { return new Symbol(Tokens.TIMES); }
<YYINITIAL> "**"  { return new Symbol(Tokens.OPOCLUSION); }
<YYINITIAL> "+"   { return new Symbol(Tokens.PLUS); }
<YYINITIAL> "++"  { return new Symbol(Tokens.PLUSPLUS); }
<YYINITIAL> "-"   { return new Symbol(Tokens.MINUS); }
<YYINITIAL> "?"   { return new Symbol(Tokens.QUESTION); }
<YYINITIAL> "|"   { return new Symbol(Tokens.PIPE); }
<YYINITIAL> "."   { return new Symbol(Tokens.DOT); }
<YYINITIAL> "("   { return new Symbol(Tokens.LPAREN); }
<YYINITIAL> ")"   { return new Symbol(Tokens.RPAREN); }
<YYINITIAL> "{"   { return new Symbol(Tokens.LCURLYBRACKET); }
<YYINITIAL> "}"   { return new Symbol(Tokens.RCURLYBRACKET); }
<YYINITIAL> "["   { return new Symbol(Tokens.LBRACKET); }
<YYINITIAL> "]"   { return new Symbol(Tokens.RBRACKET); }
<YYINITIAL> \\    { return new Symbol(Tokens.BACKSLASH); }
<YYINITIAL> "/"   { return new Symbol(Tokens.SLASH); }
<YYINITIAL> "::=" { return new Symbol(Tokens.IMPLICATION); }
<YYINITIAL> "!="  { return new Symbol(Tokens.NOTEQUAL); }
<YYINITIAL> "_"   { return new Symbol(Tokens.DONTCARE); }
<YYINITIAL> {charet} { return new Symbol(Tokens.CHARSET,yytext()); }
<YYINITIAL> [A-Z]([0-9a-zA-Z_]*[A-za-z_])?
                {Module currentModule = Specification.getInstance().getCurrentModule();
                 Identifier id = null;
                 if(currentModule != null)
                        id = currentModule.lookupModule(yytext(),
                                                new ArrayList<Module>());

                 if (id == null)
                {
                        id = new Identifier(yytext(), new UndefinedIdentifier(),currentModule);
                        if(currentModule != null)
                                currentModule.addNewId(yytext(),id);
                }

        return new Symbol(Tokens.DOMAINID,id); }
```

```
<YYINITIAL> {id} { return checkChildren(yytext()); }
<YYINITIAL> {id}("+-")  { return new Symbol(Tokens.IDPLUSSEPARATOR,yytext()); }
<YYINITIAL> {id}("*-")  { return new Symbol(Tokens.IDTIMESSEPARATOR,yytext()); }
<YYINITIAL> {id}{plusOrTimes}+
                        { return new Symbol(Tokens.IDDECORATED,yytext()); }
<YYINITIAL> {intNum} { return new Symbol(Tokens.INTEGER, new Integer(yytext())); }
<YYINITIAL> {longNum}  { return new Symbol(Tokens.LONG, new Long(yytext())); }
<YYINITIAL> {doubleNun}  { return new Symbol(Tokens.DOUBLE, new
Double(yytext())); }
<YYINITIAL> {doubleNun}("f"|"F")  { return new Symbol(Tokens.FLOAT, new
Float(yytext())); }
<YYINITIAL> [\'][A-Z][\']   { /*4*/ return new Symbol(Tokens.CHAR, new
Character(yytext().charAt(1))); }
<YYINITIAL> {quoted}  { return new Symbol(Tokens.QUOTEDSTRING,yytext()); }
<YYINITIAL> {quoted}"+"
                { return new Symbol(Tokens.QUOTEDPLUSDECORATED,yytext()); }
<YYINITIAL> {quoted}"*"
                { return new Symbol(Tokens.QUOTEDTIMESDECORATED,yytext()); }
<YYINITIAL> "+-"{quoted}
                { return new Symbol(Tokens.QUOTEDPLUSSEPARATOR,yytext()); }
<YYINITIAL> "*-"{quoted}
                { return new Symbol(Tokens.QUOTEDTIMESSEPARATOR,yytext()); }
<YYINITIAL> [ \n\t\r\f]  { /* ignore white space. */ }
<YYINITIAL>"/*" { yybegin(COMMENTS); }
<COMMENTS> [^\n] { /* ignore comments */ }
<COMMENTS> [\n]  { /* ignore comments */}
<COMMENTS> "*/" { yybegin(YYINITIAL); }
<YYINITIAL> "//"    { yybegin(COMMENT); }
<COMMENT>  [^\n]  { /* ignore comments */ }
<COMMENT>  [\n]  { yybegin(YYINITIAL);}
.                       { System.err.println("Illegal character: "+yytext());
                          System.exit(1); }
```

6.3 Arquivo Syntactic.cup (partes alteradas)

```
...
terminal
        MODULE,
        END,
        IMPORT,
        PUBLIC,
        PRIVATE,
        ELEMENT,
        TOKEN,
        IS,
        QUOTEDSTRING,
        SEMI,
        COLON,
        EQUAL,
        PIPE,
        COMPLEMENT,
```

PLUS,
TIMES,
QUESTION,
LPAREN,
RPAREN,
LBRACKET,
RBRACKET,
IGNORE,
DOT,
CHARSET,
EXTEND,
STARTSYMBOL,
IMPLICATION,
WITH,
SYNTACTIC,
COMMA,
ARROW,
LCURLYBRACKET,
RCURLYBRACKET,
FUNCTION,
TRUE,
FALSE,
DONTCARE,
IF,
THEN,
ELSE,
CASE,
OF,
LET,
IN,
WHERE,
BACKSLASH,
MINUS,
NOTEQUAL,
LT,
LTE,
GT,
GTE,
SHIFT_R1,
SHIFT_R2,
SHIFT_L,
SLASH,
MOD,
BWAND,
TILDE,
PLUSPLUS,
OR,
AND,
NOT,
EVALUATE,

```
        LEFTARROW,
        QUOTEDPLUSDECORATED,
        QUOTEDTIMESDECORATED,
        QUOTEDPLUSSEPARATOR,
        QUOTEDTIMESSEPARATOR,
        IDDECORATED,
        IDTIMESSEPARATOR,
        IDPLUSSEPARATOR,
        EMPTY,
        EVALUATION,
        PREPROC,
        INPUT,
        OUTPUT,
        APPEND,
        STDIN,
        STDOUT,
        CONTEXT,
        PERSISTENT,
        EPHEMERAL,
        MATCH,
        AT,
        FROM,
        OPOCLUSION,
         // Transformacoes
        CONTAINS,
        EXTENSION,
        TRANSFORMATION,
        SIGNATURE,
        DEFAULT,
        REPLACE,
        REDEFINE,
        BY;


terminal Integer INTEGER;
terminal Float FLOAT;
terminal Double DOUBLE;
terminal Long LONG;
terminal Identifier ID;
terminal Identifier DOMAINID;
terminal String CHAR;

non terminal module,
         importsSection,
                        domainDefinition,
                        functionDeclaration,
                        functionDefinition,
                        visibility,
                        tokenDefinition,
                        tokelemDefinition,
```

ignoreDefinition,
regularExpression,
lexemeDefinition,
regexpConcatenation,
regexpUnary,
regexpPrimary,
expression,
variableDeclaration,
variableExtension,
ruleRhsAUX,
rhsDefinition,
ruleRhs,
absRhsDefinition,
abstractRuleDefinition,
ignoreIndirectionDefinition,
grammarConstituent,
grammarTerm,
grammarUnit,
grammarTerminal,
absGrammarConstituent,
absGrammarTerm,
synDomDeclaration,
synDomDeclarationAux,
declaration,
declarationWithVisibility,
declarationWithoutVisibility,
absGrammarTerminal,
tokenExtension,
tokenElemExtension,
importsSectionAux,
moduleHeader,
identifier,
listDecoratedId,
qualifiedId,
idQualification,
domainIdentifier,
moduleEnd,
semDomDeclaration,
domainExp,
domainExpAux,
domainExpAux1,
unionDomainExp,
functionDomainExp,
instDomainExp,
listDomainExp,
primaryDomainExp,
enumDomainExp,
enumerands,
domainIdentifierExp,
tupleDomainExp1,

tupleOrIdDomainExp,
tupleDomainExp2,
tupleDomainExp1Aux,
tupleDomainExp1Aux2,
visibAux,
tupleOrIdDomainExpAux,
functionParams,
pattern,
headTailPattern,
primaryPattern,
literalValue,
signedNumberPattern,
idPattern,
tuplePattern,
listPattern,
nodePattern,
booleanValue,
operation,
patternMatchingOperation,
idQualification2,
patternAux,
patternAux1,
nodePatternElement,
compoundExpression,
ifExpression,
caseExpression,
letExpression,
whereExpression,
lambdaAbstraction,
primaryExpression,
evaluateExpression,
funUpdateExpression,
listAggregate,
caseClauses,
caseClause,
letClauses,
letClause,
patternPlusList,
createNewLevel,
deleteNewLevel,
expressionList,
expressionList1,
andOperation,
relOperation,
additiveOperation,
multOperation,
unaryOperation,
functionApplication,
evaluatePart,
funCompositionOperation,

listAppendOperation,
listConsOperation,
orOperation,
updateClauseList,
updateClause,
idOrTupleAggregate,
qualifiedIdDec,
qualifiedIdPLUSSeparator,
qualifiedIdTIMESSeparator,
nodePatternElementList,
inputSpec,
inputSpecAux,
inputDeclaration,
outputSpec,
outputSpecAux,
outputDeclaration,
contextDomainExp,
contextInfoDefList,
contextInfoDef,
persistency,
contextExpansion,
contextAggregate,
infoAssignmentList,
matchExpression,
recordPattern,
farPattern,
contextInfoPatternP,
contextInfoPatternList,
contextInfoPattern,
infoAssignmentListAux,
infoAssignment,
domainExtension,
optDefSyntacticDomain,
defSyntacticDomain,
synDomDefList,
synDomainDef,
synDomainElement,
synDomainElementList,
// transformacoes
extensionHeader,
containsSection,
transformationModule,
transfSection,
transfSectionList,
transfHeader,
signatureClause,
signatureClauseList,
labelledTypeList,
lTConstituent,
defaultClause,

```
                        defaultClauseList,
                        replaceClause,
                        replaceClauseList,
                        redefineClauseList,
                        patternStar
                        ;


module ::= moduleHeader:m importsSection:imports declaration:decs moduleEnd
        {:
                // Setar modulos importados(imports) e declarações(decs) no módulo m
        :}

        // Transformation
          | transformationModule
          | extensionHeader containsSection moduleEnd
          {:
                //ttt
                if(Notus.debug)
                        System.out.println("reduzi module de extensionHeader");
          :};

moduleEnd ::= END
                | ;

importsSection ::= IMPORT importsSectionAux SEMI
                | {: :};

// Transformation
containsSection ::= CONTAINS importsSectionAux SEMI
                | {: :};

transformationModule ::= transfHeader importsSection transfSectionList END;

transfSectionList ::= transfSectionList:list transfSection:sec
                        {::}
                        | ;


transfHeader ::= TRANSFORMATION domainIdentifier:id
                {:
                        String thisId = ((ArrayList<Identifier>)id).get(0).getName();
                        String previousId =
                                Specification.getInstance().
                                        getCurrentTransformation().getId().
                                        getName();
                        if (thisId.compareTo(previousId) != 0)

                        notus.Error.printError(TransformerMessages.DIFFERENT_IDS);
                :};
```

```
transfSection ::= SIGNATURE signatureClauseList:sig SEMI
                {:
                        Transformer transf = Specification.getInstance().
                                                        getCurrentTransformation();
                        transf.addSignature((TSignature)sig);

                        RESULT = sig;
                :}
                | DEFAULT defaultClauseList:def SEMI
                {:
                        Transformer transf = Specification.getInstance().
                                                        getCurrentTransformation();
                        transf.addDefault((TDefault)def);

                        RESULT = def;
                :}
                | REPLACE replaceClauseList:rep SEMI
                {:
                        Transformer transf = Specification.getInstance().
                                                        getCurrentTransformation();
                        transf.addReplace((TReplace)rep);

                        RESULT = rep;
                :}
                | REDEFINE redefineClauseList:red SEMI
                {:
                        Transformer transf = Specification.getInstance().
                                                        getCurrentTransformation();
                        transf.addRedefine((TRedefine)red);

                        RESULT = red;
                :};

signatureClauseList ::= signatureClause: sig
                        {:
                                RESULT = sig;
                        :};

signatureClause ::= ID:f COLON labelledTypeList: list
                {:
                        //ttt
                        if(Notus.debug)
                                System.out.println("reduzi signatureClause");

                        TSignature sig = (TSignature)list;
                        sig.defineFunction(f);

                RESULT = sig;
        :};
```

```
labelledTypeList ::= lTConstituent:lTConst
                        {:
                                //ttt
                                if(Notus.debug)
                                        System.out.println("reduzi lTConstituent
                                                                de labelledTypeList");

                                 // Criar TSignature (primeira reducao)
                                TSignature sig = new TSignature();
                                sig.addConstituent(lTConst);

                                RESULT = sig;
                        :}
                        | labelledTypeList:list ARROW lTConstituent:lTConst
                        {:
                                //ttt
                                if(Notus.debug)
                                        System.out.println("reduzi labelledTypeList
                                                ARROW lTConstituent de labelledTypeList");


                                TSignature sig = (TSignature)list;
                                sig.addConstituent(lTConst);

                                RESULT = sig;

                        :};

lTConstituent ::= instDomainExp:l
                {:
                        //ttt
                        if(Notus.debug)
                                System.out.println("reduzi instDomainExp de lTConstituent");

                        RESULT = l;
                :}
                | ID:label COLON instDomainExp:l
                {:
                        //ttt
                        if(Notus.debug)
                                System.out.println("reduzi ID COLON
                                                        instDomainExp de lTConstituent");

                        RESULT =
                                SyntacticAux.createLabelledType((DomainExpression)l, label);
                :};

defaultClauseList ::= defaultClause:def
                        {:
```

```
                        RESULT = def;
                    :};

defaultClause  ::= ID:label EQUAL expression:exp
                {:
                        RESULT = SyntacticAux.createDefaultClause(label, (Expression)exp);
                :};

replaceClauseList ::= replaceClause:rep
                    {:
                        RESULT = rep;
                    :};

replaceClause  ::= ID:f functionParams:p BY expression:exp
                {:
                        RESULT = SyntacticAux.createReplaceClause((Identifier)f,
                                        (ArrayList<FunctionPattern>)p,
                                        (Expression)exp);
                :};

redefineClauseList ::= ID:f functionParams:p BY expression:exp
                    {:
                        RESULT = SyntacticAux.createRedefineClause((Identifier)f,
                                        (ArrayList<Pattern>)p,
                                        (Expression)exp);
                    :};
```

6.4 Classe Transformation

```
package transformers;

import java.util.ArrayList;

import semantic.ast.Expression;
import notus.symbolTable.Identifier;

public class Transformer {

   private Identifier id;
   private Identifier packageId;
   private Identifier previousSpecification;

   private ArrayList<TSignature> signatures;
   private ArrayList<TDefault> defaults;
   private ArrayList<TReplace> replaces;
   private ArrayList<TRedefine> redefines;

   public Transformer(Identifier id, Identifier packageId, Identifier previousSpecification) {
      this.id = id;
      this.packageId = packageId;
      this.previousSpecification = previousSpecification;
```

```
    signatures = new ArrayList<TSignature>();
    defaults = new ArrayList<TDefault>();
    replaces = new ArrayList<TReplace>();
    redefines = new ArrayList<TRedefine>();
}

public void addDefault(TDefault tdefault) {
    this.defaults.add(tdefault);
}

public void addSignature(TSignature tsignature) {
    this.signatures.add(tsignature);
}

public void addReplace(TReplace treplace)
{
    this.replaces.add(treplace);
}

public void addRedefine(TRedefine tredefine)
{
    this.redefines.add(tredefine);
}

public Identifier getId() {
    return id;
}

public Identifier getPackageId() {
    return packageId;
}

public Identifier getPreviousSpecification() {
    return previousSpecification;
}

public ArrayList<TSignature> getSignatures() {
    return signatures;
}

public Expression getExpressionDefault(Identifier label)
{
    for (TDefault def : defaults)
    {
        // Achou o label?
        if (def.getLabel().getName().compareTo(label.getName()) == 0)
            return def.getExpression();
    }
```

```
        return null;
    }

    public ArrayList<TReplace> getReplaces() {
        return replaces;
    }

    public ArrayList<TRedefine> getRedefines() {
        return redefines;
    }
}
```

6.5 Classe TLabelledType

```
package transformers;

import notus.symbolTable.Identifier;
import semantic.ast.DomainExpression;
import semantic.ast.FunctionDomainExpComp;

// Classe criada para dar suporte a cláusula labelled-type dos transformadores.
public class TLabelledType {
    DomainExpression dExp;
    Identifier label;

    public TLabelledType(DomainExpression dExp, Identifier label)
    {
        this.dExp = dExp;
        this.label = label;
    }

    public DomainExpression getDExp() {
        return dExp;
    }

    public Identifier getLabel() {
        return label;
    }

}
```

6.6 Classe TDefault

```
package transformers;

import notus.symbolTable.Identifier;
import semantic.ast.Expression;

// Representa a clausula default das transformacoes
// que possuem sao da forma label = identifier
public class TDefault
```

```
{
   Identifier label;
   Expression exp;

   public TDefault(Identifier label, Expression exp)
   {
      this.label = label;
      this.exp = exp;
   }

   public Expression getExpression()
   {
      return exp;
   }

   public Identifier getLabel()
   {
      return label;
   }
}
```

6.7 Classe TSignature

```
package transformers;

import java.util.ArrayList;

import lexGen.LexGenerator;
import lexical.ast.ExpIdentifier;

import semantic.ast.DomainExpression;
import semantic.ast.Expression;
import semantic.ast.FunctionApplication;
import semantic.ast.FunctionDeclaration;
import semantic.ast.FunctionDefinition;
import semantic.ast.FunctionDomainExpComp;
import semantic.ast.FunctionPattern;
import semantic.ast.IdPattern;
import semantic.ast.LambdaAbstraction;
import semantic.ast.ListAggregate;
import semantic.ast.OperationOperand;
import semantic.semGen.semValue.SemDomValueType;
import syntactic.SyntacticAux;

import notus.Module;
import notus.Specification;
import notus.symbolTable.Identifier;
import notus.symbolTable.UndefinedIdentifier;

public class TSignature
{
```

```
// Identificador da funcao a que o signature se aplica.
// Foi mantido como um id porque no ponto em que o signature
// e encontrado FunctionDeclaration ainda nao foi criado.
private Identifier functionId;

// Numero de LabelledTypes inseridos. Indica quantos paramentros
// serao acrescentados pelo Signature.
private int newParametersCount;

private ArrayList<Object> constituents;

public TSignature()
{
   constituents = new ArrayList<Object>();
}

public void defineFunction(Identifier functionId)
{
   this.functionId = functionId;
}

public void addConstituent(Object constituent)
{
   constituents.add(constituent);

   if (constituent instanceof TLabelledType)
      newParametersCount++;
}

/*
public int getNewParametersCount()
{
   return newParametersCount;
}
*/

public Identifier getFunctionId()
{
   return functionId;
}

public void changeFunctionDefinition(FunctionDefinition funDef)
{
   ArrayList<FunctionPattern> patternList = funDef.getPatternList();

   int pos = 0;

   // Percorre os padroes da funcao adicionando
   // os novos parametros
   for (Object cons : constituents)
```

```
        {
            // Novo parametro?
            if (cons instanceof TLabelledType)
            {
                TLabelledType labelled = (TLabelledType)cons;

                ExpIdentifier newPar = new ExpIdentifier(labelled.getLabel());

                patternList.add(pos, newPar);
            }

            // Avanca na lista de parametros
            pos++;
        }
    }

    public void changeFunctionDeclaration(FunctionDeclaration funDec)
    {
        DomainExpression exp = funDec.getDomainExp();
        DomainExpression prev = exp;

        // Percorre os parametros declarados da funcao
        // acrescentando os novos
        for (Object cons : constituents)
        {
            // Novo parametro?
            if (cons instanceof TLabelledType)
            {
                TLabelledType labelled = (TLabelledType)cons;

                // Cria o novo elemento da lista de DomainExpression
                // da declaracao da funcao:
                // DomainExpression da signature e o left
                // DomainExpression anterior passa a ser o right
                // e o Module e o modulo de declaracao da funcao
                FunctionDomainExpComp r = new FunctionDomainExpComp(labelled.getDExp(),
exp, funDec.getModule());

                // Alterar o anterior
                if ( !(prev instanceof FunctionDomainExpComp))
                {
                    FunctionDomainExpComp aux = new FunctionDomainExpComp(prev, r,
funDec.getModule());
                    prev = aux;
                }

                ((FunctionDomainExpComp)prev).setRight(r);
                // Mexeu na raiz (primeiro DomainExpression de funDec)?
                if (exp == funDec.getDomainExp())
                    funDec.setDomainExp(prev);
```

```
            prev = r;
         }
         else
         {
            // Avanca na lista
            if (exp instanceof FunctionDomainExpComp)
            {
               prev = exp;
               exp = ((FunctionDomainExpComp)exp).getRight();
            }
         }
      }
   }

   public LambdaAbstraction createLambdaAbstraction(FunctionDeclaration currentFunction)
   {
      Specification specification = Specification.getInstance();
      String fullFuncTransName =
((FunctionDeclaration)functionId.getRole()).getFullName();

      // Usa o modulo da declaracao da funcao para criar os ids
      // expressao lambda
      Module modIds = currentFunction.getModule();

      // Lista de variaveis da expressao lambda
      ArrayList<FunctionPattern> patternList = new ArrayList<FunctionPattern>();

      // Preenche a lista acima
      LexGenerator lexGen = LexGenerator.getInstance();
      for (int i = 0; i < newParametersCount; i++)
      {
         String cont = lexGen.getGlobalCont();
         String idName = "___lamb_" + Integer.toString(i) + cont;
         Identifier id = new Identifier(idName, null, modIds);
         modIds.addNewId(id.getName(), id);
         ExpIdentifier expId = new ExpIdentifier(id);
         /* TODO: TESTAR */
         //expId.setType(/* ... inteiro ...*/);
         patternList.add(expId);
      }

      // Cria a expressao que sera utilizada na expressao lambda
      OperationOperand prevFunction = new ExpIdentifier(functionId);
      Expression lambdaExpression = null;
      // Controle de variaveis lambda utilizadas
      int j = 0;
      // Percorre os elementos do signature (excluindo o retorno)
      for (int i = 0; i < constituents.size() - 1; i++)
      {
```

```
            Object cons = constituents.get(i);
            OperationOperand parameter;

            // E um novo parametro?
            if (cons instanceof TLabelledType)
            {
               TLabelledType lt = (TLabelledType)cons;

               // Deve usar a expressao default ou propagar
               // o parametro?
               if (currentFunction.getFullName().compareTo(fullFuncTransName) == 0)
               {
                  // Propagar o parametro (chamada recursiva)
                  parameter = new ExpIdentifier(lt.getLabel());
               }
               else
               {
                  // Usar default
                  // Obtem a expressao default
                  Expression defaultExpression =
specification.getCurrentTransformation().getExpressionDefault(lt.getLabel());
                  // Para passar expression para function application
                  // e necessario criar um ListAggregate
                  ArrayList<Expression> list = new ArrayList<Expression>();
                  list.add(defaultExpression);
                  parameter = (OperationOperand)SyntacticAux.createListAggregate(list);
               }
            }
            // Nao: usar variavel lambda
            else
            {

               parameter = (ExpIdentifier)patternList.get(j);

               // Usou uma variavel
               j++;
            }

            lambdaExpression = new FunctionApplication(prevFunction, parameter);
            prevFunction = (FunctionApplication)lambdaExpression;
         }

         // Cria a expressao lambda com a lista de parametros acima e
         // com a expressao ja modificada.
         LambdaAbstraction aux = SyntacticAux.createLambdaAbstraction(patternList,
lambdaExpression);

         // FEITO NO CHAMADOR:
         // CRIAR O PARAMETRO DA EXPRESSAO LAMBDA
         // Altera exp para ser o parametro da expressao lambda
```

```
        return aux;
    }
}
```

6.8 Classe TReplace

```
package transformers;

import java.util.ArrayList;

import lexGen.LexGenerator;
import lexical.ast.ExpIdentifier;

import notus.Module;
import notus.symbolTable.Identifier;
import semantic.ast.CaseClause;
import semantic.ast.CaseExpression;
import semantic.ast.DontCare;
import semantic.ast.Expression;
import semantic.ast.FunctionApplication;
import semantic.ast.FunctionDeclaration;
import semantic.ast.FunctionPattern;
import semantic.ast.LambdaAbstraction;
import semantic.ast.ListAggregate;
import semantic.ast.OperationOperand;
import syntactic.SyntacticAux;

// Transformers
public class TReplace
{
    private Expression expression;
    private Identifier functionId;
    private ArrayList<FunctionPattern> patternList;

    public TReplace(Identifier functionId, ArrayList<FunctionPattern> patternList, Expression
exp)
    {
        this.expression = exp;
        this.functionId = functionId;
        this.patternList = patternList;
    }

    public Identifier getFunctionId()
    {
        return functionId;
    }
```

```
  public LambdaAbstraction createCaseExpression(FunctionDeclaration currentFunction)
  {
     LexGenerator lexGen = LexGenerator.getInstance();

     // Usa o modulo da declaracao da funcao para criar os ids
     // expressao do case
     Module modIds = currentFunction.getModule();

     // Cria a expressao que sera utilizada na expressao lambda
     Expression lambdaExpression = null;

     // Cria o else de todos os padroes
     DontCare caseDontCare = new DontCare();
     // Cria a expressao que sera utilizada na expressao lambda
     OperationOperand caseDefaultFunction = new ExpIdentifier(functionId);
     OperationOperand prevFunction = caseDefaultFunction;

     // Lista com as variaveis da expressao lambda
     ArrayList<FunctionPattern> lambdaPatternList = new ArrayList<FunctionPattern>();

     ArrayList<Expression> listExp = new ArrayList<Expression>();
     for (int i = 0; i < patternList.size(); i++)
     {
        // Cria os identificadores que serao usados
        String idCase = "___replace_case_id_" + lexGen.getGlobalCont();
        Identifier id = new Identifier(idCase, null, modIds);
        modIds.addNewId(id.getName(), id);
        ExpIdentifier expId = new ExpIdentifier(id);
        lambdaPatternList.add(expId);

        // Cria a expressao default do case, ie,
        // a funcao original aplicada a seus argumentos
        // agora sob a expressao lambda
        caseDefaultFunction = new FunctionApplication(prevFunction, expId);
        prevFunction = (FunctionApplication)caseDefaultFunction;

        // Preenche lista dos elementos das expressoes dos case
        listExp.add(expId);

        // Cria a cadeia de FunctionApplication da expressao lambda
        lambdaExpression = new FunctionApplication(prevFunction, expId);
        prevFunction = (FunctionApplication)lambdaExpression;
     }

     // Cria o padrao default dos cases
     CaseClause caseDefault = SyntacticAux.createCaseClause(caseDontCare,
caseDefaultFunction);

     CaseClause caseOk;
```

```
        // Cria os cases
        CaseExpression caseExpression = null;
        Expression prevCase = this.expression;
        for (int i = 0; i < listExp.size(); i++)
        {
           Expression ei = listExp.get(i);

           // Cria o caso em que deve passar p/ o proximo
           // case, ie, o padrao foi reconhecido
           caseOk = SyntacticAux.createCaseClause(this.patternList.get(i), prevCase);

           // Clausulas dos cases
           ArrayList<CaseClause> caseClauses = new ArrayList<CaseClause>();
           caseClauses.add(caseOk);
           caseClauses.add(caseDefault);

           caseExpression = SyntacticAux.createCaseExpression(ei, caseClauses);
        }


        // Cria a expressao lambda que contem os cases
        LambdaAbstraction lambda =
SyntacticAux.createLambdaAbstraction(lambdaPatternList, caseExpression);

        return lambda;
    }
}
```

6.9 Classe TRedefine

```
package transformers;

import java.util.ArrayList;
import java.util.List;

import notus.Error;
import notus.Notus;
import notus.symbolTable.Identifier;
import semantic.ast.Expression;
import semantic.ast.FunctionDeclaration;
import semantic.ast.FunctionDefinition;
import semantic.ast.FunctionPattern;
import semantic.semGen.SemGenerator;
import semantic.semGen.semValue.SemDomValueType;
import syntactic.SyntacticAux;
import syntactic.grammarSimplification.NotusGraph;

public class TRedefine
{

    // Definicao que sera utilizada para fazer o redefine
```

```
   private FunctionDefinition newDef;

   public TRedefine(Identifier functionId, ArrayList<FunctionPattern> patternList,
Expression exp)
   {
      newDef = SyntacticAux.createFunctionDefinition(null, functionId, patternList, exp);

      /*
      this.expression = exp;
      this.functionId = functionId;
      this.patternList = patternList;
      */
   }

   public void applyRedefine(FunctionDeclaration funDec)
   {
      SemGenerator semg = SemGenerator.getInstance();

      // Nome completo da funcao do redefine
      String fullRedName =
((FunctionDeclaration)newDef.getFunctionId().getRole()).getFullName();

      // Nomes completos diferentes?
      if (fullRedName.compareTo(funDec.getFullName()) != 0)
      {
         // Nada a fazer
         return;
      }

      // Adianta a geracao do cabecalho da funcao para comparar os padroes
      semg.createFunHeaderAux(funDec);

      FunctionDefinition funDef;
      List<FunctionDefinition> funDefs = funDec.getFunDefs();
      semg.genFunParams(funDec, newDef);

      // Percorre a definicao das funcoes
      for (int i = 0; i < funDefs.size(); i++)
      {
         // Proxima definicao a verificar
         funDef = funDec.getFunDefs().get(i);

         // Ao criar newDef o compilador o insere na lista de
         // definicoes, o que nao esta certo, solucao:
         // Deixar inserir e remove-lo
         if (funDef.equals(newDef))
         {
            funDefs.remove(i);
            continue;
         }
```

```
            // Cria os parametros para realizar a verificacao
            semg.genFunParams(funDec, funDef);

            // Compara os padroes
            //Compara conforme SemGenerator
            int[] matchingResult = semg.matchingFunDefParam(funDef.getFunParams(),
                                                    newDef.getFunParams());
            int state = semg.runFunDefConflictedAFD(matchingResult);

            // Encontrado? Transformar?
            if (state == SemGenerator.EQUALS)
            {
               funDefs.remove(i);
               funDefs.add(i, newDef);
            }
         }
      }
   }
```

BIBLIOGRAFIA CITADA

Tirelo, F., Bigonha, R. S., Saraiva, J. A. B. ; Disentangling Denotational Semantics Definitions; SBLP, 2008;

Gordon, M. J. C.; The Denotational Description of Programming: Languages - An Introduction; Springer-Verlag, 1979 (2nd revised edition).

Lex; The LEX & YACC Page, All about *Lex, Yacc*, Flex, and Bison: Overview, Online Documentation, Papers, Tools, Pointers; *dinosaur.compilertools.net*.

Yacc; The LEX & YACC Page, All about *Lex, Yacc*, Flex, and Bison: Overview, Online Documentation, Papers, Tools, Pointers; *dinosaur.compilertools.net*.

Soares, T. C. A.; Compilação de Semântica Denotacional Modular; Dissertação de Mestrado; DCC, UFMG, 2007.