

Universidade do Minho
Conselho de Cursos de Engenharia
Licenciatura em Engenharia Informática

Disciplina de LI IV - Projecto

Ano Lectivo de 2007/08



escola de engenharia



departamento de
informática

Calculador Pointfree

João Moura Barbosa, Mário Ulisses Costa

Julho, 2008

Data de Recepção	
Responsável	
Avaliação	
Observações	

Calculador Pointfree

João Moura Barbosa

Mário Ulisses Costa

Julho, 2008

Resumo

O objectivo do projecto é a implementação de uma ferramenta capaz de auxiliar o cálculo de programas em *pointfree*. Foi dada principal atenção à usabilidade, tentando oferecer uma interface amigável e de fácil utilização.

Ficou implementado o cálculo de funções não recursivas e todos os combinadores conhecidos para estas: composição, produtos e co-produtos.

Área de Aplicação: Cálculo de Programas.

Palavras-Chave: *Cálculo Pointfree*, Cálculo de Programas, JAVA e ANTRL.

Índice

Resumo	i
Índice	iii
Índice de Figuras	iv
Índice de Tabelas	v
1 Introdução	1
1.1 Contextualização do Caso de Estudo	1
1.2 Motivação e Objectivos	2
1.3 Estrutura do Relatório	2
2 Representação	4
2.1 Parsing	4
2.2 Expressões	6
2.3 Regras	7
2.4 Provas	8
3 Calculador	10
3.1 Path	10
3.2 Aplicação de regras	11
4 Apresentação	13
4.1 Estrutura	13
4.2 Navegação	14
4.3 <i>Highlighting</i>	16
4.3.1 <i>Character invisível</i>	16
5 Conclusões e Trabalho Futuro	17

Bibliografia	17
Referências WWW	19
Lista de Acrónimos	20
A Anexos	21
A.1 Funções JAVA para os métodos de navegação	21

Índice de Figuras

2.1	Gramática para a linguagem <i>pointfree</i>	5
2.2	Representação em objectos <i>JAVA</i> da expressão 2.1	7
2.3	Ficheiro de regras.	7
2.4	Representação abstracta da <i>Univsertal</i> – +.	8
4.1	Interface	14

Índice de Tabelas

2.1	Representação dos combinadores <i>pointfree</i>	5
-----	---	---

The idea of producing programs by calculation, that is to say, that of calculating efficient programs out of abstract.

J. N. Oliveira

1

Introdução

1.1 Contextualização do Caso de Estudo

“ *O estilo de programação pointfree, ao contrário do estilo pointwise, caracteriza-se pela ausência de variáveis na definição de funções. No estilo pointwise uma função é definida especificando directamente o seu comportamento num determinado ponto do domínio, usualmente denotado por uma variável ou um padrão envolvendo variáveis.*

No estilo pointfree uma função é definida por combinação de outras funções mais simples usando um conjunto limitado de combinadores. A escolha destes combinadores é ditada pelo poder das leis de cálculo que lhes estão associadas. Este facto implica que, normalmente, seja mais fácil demonstrar propriedades sobre programas escritos neste estilo. [3]

O *Cálculo Pointfree* reduz-se, então, à escolha e aplicação acertada de regras formalmente derivadas. A tarefa, apesar de ser relativamente simples, torna-se entediante e repetitiva para provas de tamanho considerável. Além disso, exige concentração na reescrita da prova depois da aplicação de uma regra.

Como podemos ver no exemplo 1.1, a aplicação da *Fusion* $- +$ à expressão obrigou à reescrita da prova e ainda à repetição da *inl* ∇ *inr*.

À medida que as provas vão aumentando, a probabilidade do calculador cometer uma gafe por distração é elevada e, depois de algum traquejo no cálculo, este facto torna-se

demasiado contraprodutivo.

$$(inl \nabla inr) \circ (id \nabla id) = id \tag{1.1}$$

$$\Leftrightarrow \{Fusion - +\}$$

$$((inl \nabla inr) \circ id \nabla (inl \nabla inr) \circ id) = id$$

1.2 Motivação e Objectivos

O intuito deste projecto é precisamente facilitar o *Cálculo Pointfree*, retirando ao utilizador o trabalho repetitivo e demeritório.

A possibilidade de construir uma ferramenta para auxiliar o estudo de uma das disciplinas preferidas pelos autores foi, sem dúvida, uma grande motivação, no entanto, o motivo preponderante foi a esperança de aumentar o nível de conhecimentos na área.

1.3 Estrutura do Relatório

Este documento foi pensado em quatro grandes secções. São elas:

Capítulo 1º O leitor é introduzido ao tema abordado, o *Cálculo Pointfree*, e é justificada a utilidade do projecto.

Capítulo 2º É discutida com relativa profundidade a forma escolhida para representar os elementos do *Cálculo Pointfree*: *expressões, regras e provas*. É também brevemente explicada a gramática utilizada no *parsing* de expressões.

Capítulo 3º Todo o *cálculo* propriamente dito, i.e., a aplicação de regras, as sucessivas substituições e todos os elementos auxiliares como o *Path* são abordados com detalhe nesta secção.

Capítulo 4º Capítulo dedicado à interacção do utilizador com a ferramenta. É abordada a navegação numa expressão e o *interface* disponibilizado.

Capítulo 5º No capítulo final são tecidos alguns comentários relativos ao trabalho efectuado

e motivação para trabalho futuro.

Language is a process of free creation; its laws and principles are fixed, but the manner in which the principles of generation are used is free and infinitely varied. Even the interpretation and use of words involves a process of free creation.

Noam Chomsky

2

Representação

Conteúdo

2.1 Parsing	4
2.2 Expressões	6
2.3 Regras	7
2.4 Provas	8

2.1 Parsing

A tradução de texto para estruturas de dados é um passo fulcral e, como tal, será o primeiro a ser abordado. Para linguagens relativamente complexas é imperativa a utilização de um gerador de parsers, em detrimento de uma construção manual.

A nossa escolha passou pelo **ANTLR**, por ser uma *framework* com vasta documentação e por estar bem integrada no **JAVA**.

De forma a facilitar a inserção de expressões *pointfree*, não usamos caracteres especiais de *unicode*. Em vez disso, o utilizador insere os caracteres da tabela 2.1, que são posteriormente convertidos para nos homólogos em *unicode*.

Combinador original	Combinador utilizado	Aridade
\circ	.	<i>n</i> -ário
∇	$\backslash/$	binário
Δ	$/\backslash$	binário
+	- -	binário
\times	><	binário
\wedge	&	<i>n</i> -ário
\Leftrightarrow	:=:	binário
=	=	binário

Tabela 2.1: Representação dos combinadores *pointfree*.

Como o **ANTLR** apenas reconhece gramáticas *LL*, para o *parsing* foi definida uma linguagem a partir da GIC *LL* na figura 2.1.

```

1 definicoes -> definicao definicao*
2
3 definicao  -> ID ':' ID '/' NUM '=' expressao
4           | ID ':' ID
5
6 regras    -> regra regra*
7
8 regra     -> ID ':' expressao '=' expressao
9
10
11 expressao -> apos (OPERADOR apos)?
12
13 apos      -> equiv ('.' equiv)*
14
15 equiv     -> and ('<=>' and)?
16
17 and       -> equal ('&' equal)*
18
19 equal     : factor ('=' factor)?
20
21 factor    -> ID
22           | '(' expressao ')'
23
24 NUM       -> ('0'..'9')+
25
26 ID        -> ('a'..'z' | 'A'..'Z') ('0'..'9' | 'a'..'z' | 'A'..'Z' | '+' | '-')*
27
28 OPERADOR  -> ('\|/ | '\|\' | '-|- | '><')
```

Figura 2.1: Gramática para a linguagem *pointfree*.

Apesar da teoria na definição de linguagens ultrapassar o tema do projecto, é pertinente a abordagem a alguns aspectos relativos à gramática escolhida.

A simplicidade de utilização foi, desde o início, um ponto importante e, por isso, foi dada maior precedência a alguns operadores, de forma a reduzir substancialmente o número de parênteses. A definição dos operadores \circ e \wedge como *n*-ários também veio no sentido da

redução de parênteses.

O problema mais complicado de lidar foi com o facto de a gramática ter que ser *LL*, ao contrário das que os autores estavam mais habituados: *LR*.

“ *The class of LL grammars is rich enough to cover most programming constructs, although care is needed in writing a suitable grammar for the source language. For example, no left-recursive or ambiguous grammar can be LL. A grammar G is LL if and only if whenever $A \rightarrow \alpha|\beta$ are two distinct productions of G , the following conditions hold:*

1. *For no terminal α do both α and β derive strings beginning with α .*
2. *At most one of α and β can derive the empty string.* [1]

2.2 Expressões

Definição Uma expressão foi definida como uma árvore n -ária de expressões, cujos nós são operadores infixos ou terminais (conjunto de expressões vazio). Uma expressão terminal é variável ou concreta, dependendo da existência de uma predefinição através de uma regra.

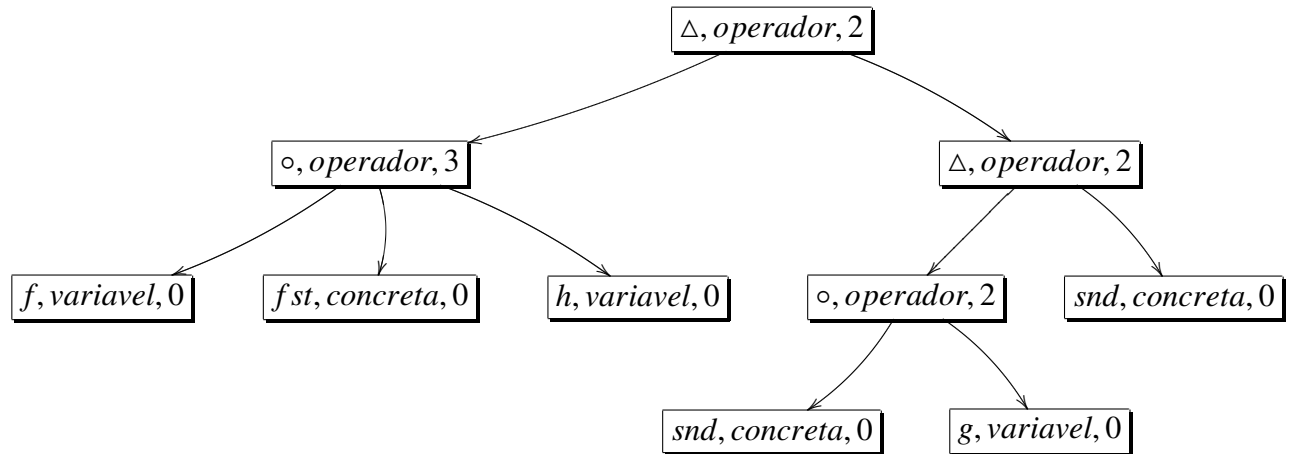
Segundo a definição recursiva de expressão, as seguintes expressões são válidas:

- $(f \circ fst \circ h) \Delta ((snd \circ g) \Delta snd)$ ou: $f \circ fst \circ h \Delta (snd \circ g \Delta snd)$
- $(id \times fst) \wedge (snd \circ snd)$ ou: $(id \times fst) \wedge snd \circ snd$

Usando *(símbolo, tipo, aridade)* para definir um Nó da árvore n -ária, podemos representar a expressão

$$f \circ fst \circ h \Delta (snd \circ g \Delta snd) \tag{2.1}$$

na árvore da figura 2.2.

Figura 2.2: Representação em objectos *JAVA* da expressão 2.1

2.3 Regras

Definição Uma regra é definida como uma equivalência de duas expressões.

As regras são carregadas a partir de um ficheiro com o padrão da figura 2.3.

```

1 Cancel-x1 : fst . (f ∧ g) = f
2 Cancel-x2 : snd . (f ∧ g) = g
3 Reflex-x : fst ∧ snd = id
4 Fusion-x : (f ∧ g) . h = f . h ∧ g . h
5 Def-x : f >> g = f . fst ∧ g . snd
6 Absor-x : (f >> g) . (h ∧ i) = f . h ∧ g . i
7 Functor-x : (f >> g) . (h >> i) = f . h >> g . i
8 Functor-id-x : id >> id = id
9 Nat-fst : fst . (f >> g) = f . fst
10 Nat-snd : snd . (f >> g) = g . snd
11 Cancel-+1 : (f ∨ g) . inl = f
12 Cancel-+2 : (f ∨ g) . inr = g
13 Reflex-+ : inl ∨ inr = id
14 Fusion-+ : f . (g ∨ h) = f . g ∨ f . h
15 Def-+ : f -|- g = inl . f ∨ inr . g
16 Absor-+ : (f ∨ g) . (h -|- i) = f . h ∨ g . i
17 Functor-+ : (f -|- g) . (h -|- i) = f . h -|- g . i
18 Functor-id-+ : id -|- id = id
19 Lei-Troca : (f ∧ g) ∨ (h ∧ i) = (f ∨ h) ∧ (g ∨ i)
20 Nat-id : f . id = f
21 Univ-x : h :=: (f ∧ g) = ((fst . h :=: f) & (snd . h :=: g))

```

Figura 2.3: Ficheiro de regras.

A regra *Universal* – + 2.2 teria, então, a representação abstracta da figura 2.4 em **JAVA**.

$$h = (f \Delta g) \Leftrightarrow ((fst \circ h = f) \wedge (snd \circ h = g)) \quad (2.2)$$

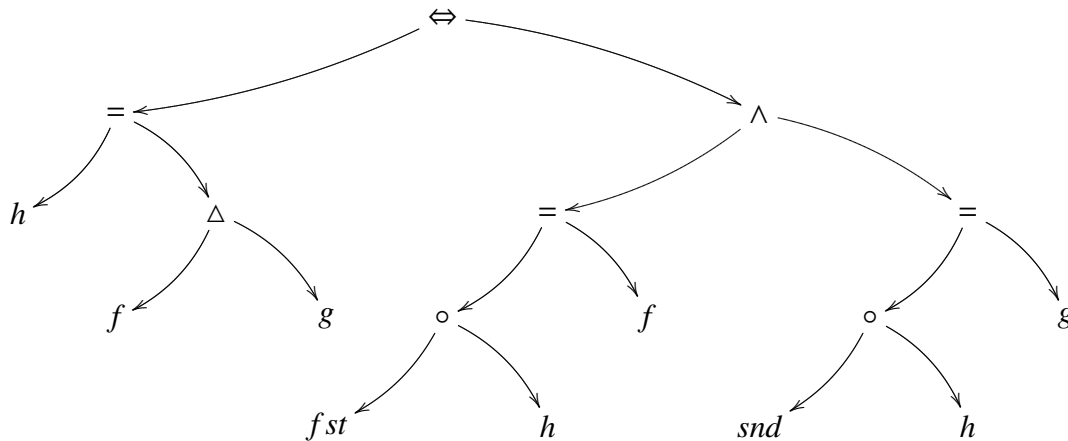


Figura 2.4: Representação abstracta da *Universal-+*.

2.4 Provas

O estado do cálculo é mantido num objecto **JAVA** (Prova), onde é guardada uma expressão sobre a qual se itera e a posição actual na mesma. A dita expressão não é mais do que o operador binário = e as expressões cuja equivalência se pretende provar, por seu lado a posição numa expressão é guardada também num objecto **JAVA** ver 3.1 (Path). Além disso, ainda são guardadas as regras previamente carregadas do ficheiro pré-definido.

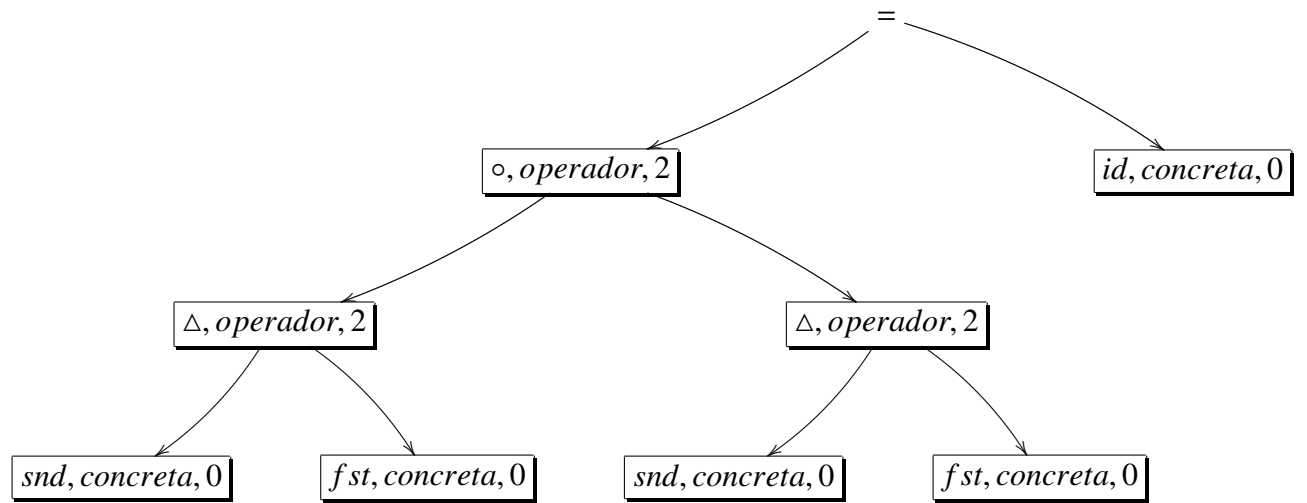
Desta forma, a prova de $swap \circ swap = id$ depois da aplicação de *Def - swap*,

$$swap \circ swap = id \quad (2.3)$$

$$\Leftrightarrow \{Def - swap\}$$

$$(snd \Delta fst) \circ (snd \Delta fst) = id$$

fica representada da seguinte maneira:



Beware of bugs in the above code; I have only proved it correct, not tried it.

Donald E. Knuth

3

Calculador

Conteúdo

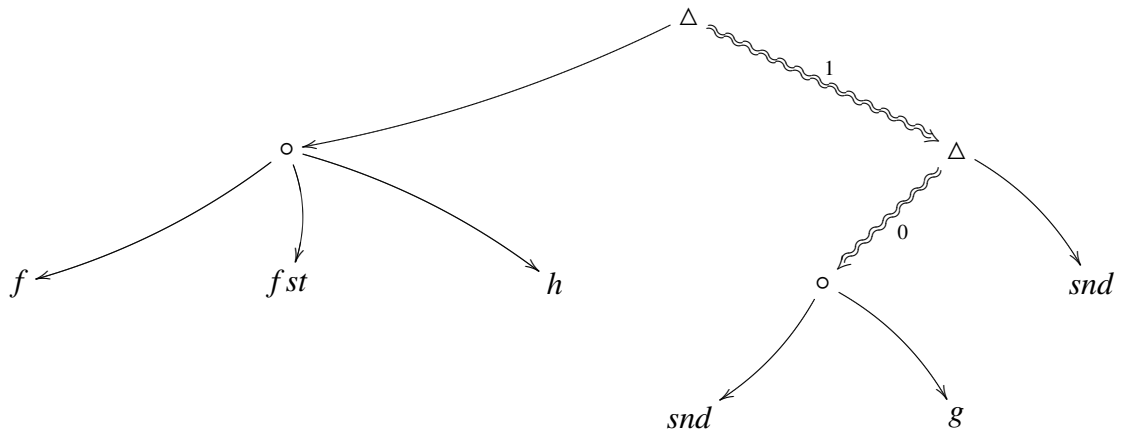
3.1 Path	10
3.2 Aplicação de regras	11

3.1 Path

Definição Um *Path* indica univocamente uma sub-expressão contida numa expressão.

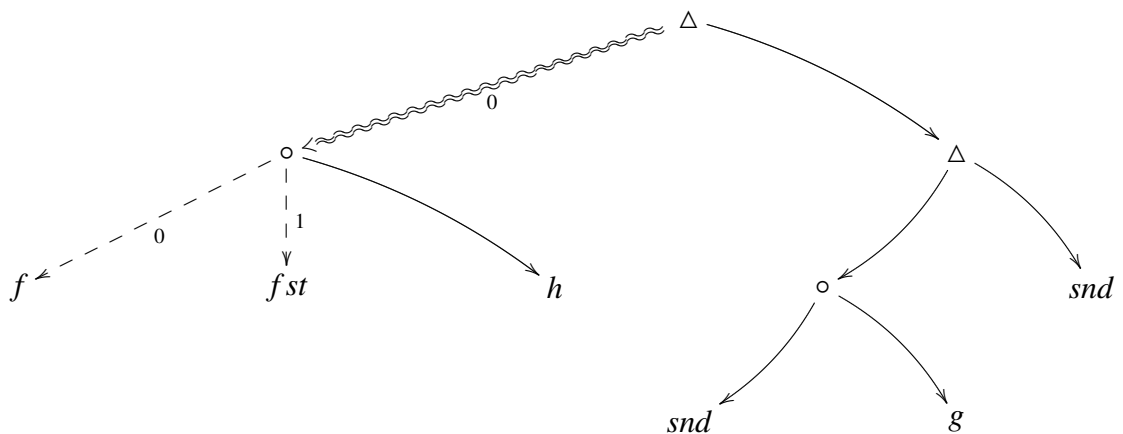
A prova de regras *pointfree* consiste em sucessivas instanciações de regras previamente calculadas. Por isso, ao ter como objectivo a automatização da substituição nas expressões, nasce a necessidade de localizar estas dentro de outras. Tendo em vista a facilitação da navegação nas expressões do mesmo nível, o *Path* foi definido contendo dois conjuntos.

O primeiro conjunto, o nível, é representado como a *stack* de passos necessários para entrar numa sub-expressão. Na árvore abaixo usou-se o path `/1/0/[]` para localizar a expressão $snd \circ g$.



Suponhamos que se pretendia seleccionar $f \circ fst$ no caso anterior. Como o nível abrange todos os ramos de uma expressão é necessária a existência de um segundo conjunto: os filhos visitados.

Na árvore abaixo usou-se, então, o path $/0/[0, 1]$ para localizar a expressão $f \circ fst$.



3.2 Aplicação de regras

O algoritmo de aplicação de regras passa por três fases distintas:

1 Selecção da expressão O utilizador escolhe a sub-expressão a substituir. Internamente é construído um *Path* e mostrada a lista de regras aplicáveis à expressão seleccionada.

2 Instânciação da regra Depois de seleccionada a regra a aplicar, é construída uma árvore resultante da aplicação da regra à sub-expressão seleccionada, i.e, todas as expressões variáveis da regra são instanciadas nas sub-expressões da prova.

*Behind all their personal
vanity women themselves
always have their impersonal
contempt.*

Friedrich Nietzsche

4

Apresentação

Conteúdo

4.1 Estrutura	13
4.2 Navegação	14
4.3 Highlighting	16
4.3.1 Caracter invisível	16

4.1 Estrutura

A camada de *interface* com o utilizador está dividida em 3 partes:

- 1 Área de texto inferior** onde o utilizador pode inserir a equação que pretende provar.
- 2 Área de texto superior** uma área de tamanho privilegiado, onde toda a prova será executada e onde o utilizador poderá navegar nas respectivas sub-expressões.
- 3 Listagem do lado direito** listagem das regras que são aplicáveis à sub-expressão seleccionada.

No componente 1 o utilizador terá que inserir equações do estilo:

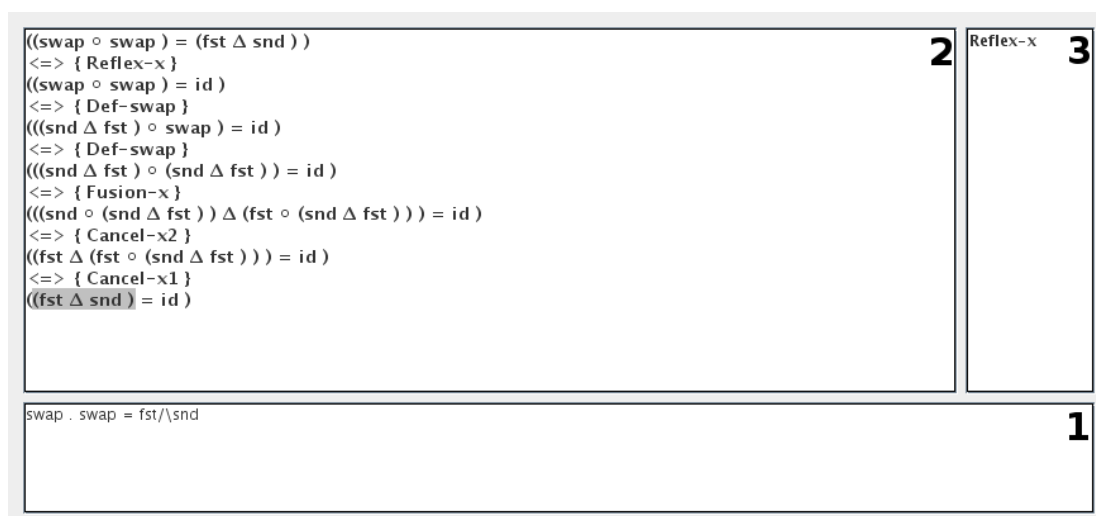


Figura 4.1: Interface

$$expressao = expressao \quad (4.1)$$

Tendo, claro, de obedecer às regras da gramática da figura 2.1.

Assumindo que foi declarado *assocr* no ficheiro de regras como:

$$Def - assocr : assocr = (fst \circ fst) \Delta (snd \times id) \quad (4.2)$$

Podemos provar que a equação 4.3 é verdadeira.

$$assocr \circ ((f \times g) \times h) = (f \times (g \times h)) \circ assocr \quad (4.3)$$

A linha é copiada para o componente 2 e é contruída a respectiva árvore n -ária que a representa.

4.2 Navegação

O utilizador tem ao seu dispôr cinco métodos para navegar nas sub-expressões, todos estes métodos ganham vida com as teclas cujos seus nomes referem. Todos estes métodos trabalham sobre um *path*, alterando-o.

Up

Através de *up* o utilizador pode subir para o nodo pai da sub-expressão que está seleccionada.

Supondo a seguinte situação:

$$assocr \circ ((f \times g) \times h) = (f \times (g \times h)) \circ assocr \quad (4.4)$$

A aplicação de *up* resultaria em:

$$assocr \circ ((f \times g) \times h) = (f \times (g \times h)) \circ assocr \quad (4.5)$$

Down

Através de *down* o utilizador pode descer na expressão para a sub-expressão mais à esquerda.

Ou seja, supondo:

$$assocr \circ ((f \times g) \times h) = (f \times (g \times h)) \circ assocr \quad (4.6)$$

A aplicação de *down* resulta em:

$$assocr \circ ((f \times g) \times h) = (f \times (g \times h)) \circ assocr \quad (4.7)$$

Left

Por sua vez, através do método *left*, o utilizador pode navegar nos filhos da sub-expressão pai da que se está a seleccionar actualmente, ou seja os irmãos.

Mais uma vez, supondo:

$$assocr \circ ((f \times g) \times h) = (f \times (g \times h)) \circ assocr \quad (4.8)$$

A aplicação de *left* resultaria em:

$$assocr \circ ((f \times g) \times h) = (f \times (g \times h)) \circ assocr \quad (4.9)$$

Right

O comportamento de *right* é análogo a *left*, tendo em conta as diferenças óbvias.

Tab

Finalmente, o método *tab*, possibilita ao utilizador a selecção de vários nodos ao mesmo nível, a partir do actualmente seleccionado.

Ou seja, assumindo que a selecção actual é:

$$\boxed{\text{assocr}} \circ ((f \times g) \times h) \circ h \quad (4.10)$$

Depois de aplicar *tab* obtem-se:

$$\boxed{\text{assocr} \circ ((f \times g) \times h)} \circ h \quad (4.11)$$

E, repetindo a operação,

$$\boxed{\text{assocr} \circ ((f \times g) \times h) \circ h} \quad (4.12)$$

4.3 Highlighting

A *Class Highlighting* é uma adaptação da apresentada no livro [4]. Esta classe recebe um componente do **JAVA** (e.g. área de texto ou listagem de conteúdos) e uma sequência de caracteres. Ao ser criado um objecto deste tipo, todas as ocorrências da frase referenciada são, na componente seleccionada, *highlighted*.

4.3.1 Character invisível

Uma expressão pode ter muitas palavras repetidas, exemplificado aqui na expressão 4.12, e como só é desejado que o *highlight* aconteça numa delas, *h* no exemplo, a *String* resultante da expressão que se pretende mostrar é ladeada por dois caracteres invisíveis. O carácter escolhido foi `\r`. Desta forma, iríamos ter um *h* sem *highlight* e outro *h* com *highlight*.

5

Conclusões e Trabalho Futuro

O principal *handicap* do projecto foi o reconhecedor da linguagem. A partir do momento em que este foi ultrapassado, o trabalho fluiu relativamente bem.

À medida que fomos avançando no terreno, novas ideias foram surgindo e ficaram, com certeza, muitas mais interessantes por implementar.

Dada a magnitude da área e o conhecimento incompleto do terreno por parte dos autores, o fim do túnel permaneceu invisível.

O trabalho futuro passa primordialmente por incorporar um sistema de verificação de tipos e cálculo para tipos recursivos. Apesar disto não vir a acontecer a tempo da avaliação final, os autores têm a esperança dos implementar futuramente com o intuito de ajudar o cálculo de programas.

Continuamos com a certeza de que era o melhor projecto da lista.

Bibliografia

- [1] Alfred V. Aho, Compilers : principles, techniques, and tools, 2nd edition Addison Wesley, 1986.
- [2] J.N. Oliveira. Program Design by Calculation, Departamento de Informática, Universidade do Minho, 2005.
- [3] A. Cunha. Cálculo de Programas: notas teórico-práticas. Departamento de Informática, Universidade do Minho, 2005.
- [4] Patrick Chan, The Java Developers Almanac, 4th edition, Prentice Hall, 2002.
- [5] http://en.wikipedia.org/wiki/LL_parser.
- [6] http://en.wikipedia.org/wiki/LR_parser.

Referências WWW

- 01 <http://www.antlr.org/>
Página principal da *framework* **ANTLR**.
- 02 http://exampledepot.com/egs/javax.swing.text/style_HiliteWords.html
Página com um exemplo de *Highlighting* de palavras em **JTextComponent**.

Lista de Acrónimos

ANTLR ANother Tool for Language Recognition

LR LR parser is a parser for context-free grammars that reads input from Left to right and produces a Rightmost derivation [6]

LL An LL parser is a top-down parser for a subset of the context-free grammars. It parses the input from Left to right, and constructs a Leftmost derivation of the sentence [5]

GIC Gramática Independente do Contexto

A

Anexos

A.1 Funções JAVA para os métodos de navegação

Up

```
1 public void up() {
2     if (!pos.getFilhos().isEmpty())
3         pos.newFilhos();
4     else if (!pos.empty())
5         pos.pop();
6 }
```

Down

```
1 public void down() {
2     if (!pos.getFilhos().isEmpty() && prova.getSubExp(pos).getAridade() > 0) {
3         int i = pos.getFilhos().get(0);
4         pos.newFilhos();
5         pos.push(i);
6     }
7 }
```

Left

```
1 public void left() {
2     if (pos.getFilhos().isEmpty())
3         pos.pushFilho(0);
4     else {
5         int i = pos.getFilhoN(0) - 1;
6         pos.newFilhos();
7         pos.pushFilho(0);
8         if (i >= 0) {
```

A.1. FUNÇÕES JAVA PARA OS MÉTODOS DE NAVEGAÇÃO

```
9         pos.newFilhos();
10        pos.pushFilho(i);
11    }
12
13    }
14 }
```

Right

```
1 public void right() {
2     int ari = this.getAridade();
3     if (pos.getFilhos().isEmpty())
4         pos.pushFilho(0);
5     else {
6         int i = pos.getFilhoN(0) + 1;
7         if (i < ari) {
8             pos.newFilhos();
9             pos.pushFilho(i);
10        }
11    }
12 }
```

Tab

```
1 public void tab() {
2     if (!pos.getFilhos().isEmpty()) {
3         int ari = this.getAridade();
4         int size = pos.getFilhos().size();
5         int i = pos.getFilhoN(size - 1) + 1;
6
7         if (i < ari)
8             pos.pushFilho(i);
9     }
10 }
```