Mestrado em Engenharia Informática

Universidade do Porto

Faculdade de Engenharia

FEUP

Testes de Programas em Haskell

Teste e Qualidade de Software

Pedro Rodrigo Caetano Strecht Ribeiro

Junho 2005

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Este relatório apresenta o resultado final do estudo e desenvolvimento de uma arquitectura de testes de programas escritos em linguagem Haskell, no âmbito do projecto da disciplina de Teste e Qualidade de Software da edição de 2004/2005 do Mestrado em Engenharia Informática da Faculdade de Engenharia da Universidade do Porto.

A arquitectura implementada visa fornecer uma plataforma para testar programas em Haskell, utilizando como base a *framework* HUnit 1.0. Como exemplo de aplicação em teste utilizou-se uma aplicação desenvolvida na mesma linguagem.

O segundo capítulo apresenta a linguagem Haskell enquadrado-a no paradigma da programação funcional e comparando-a com linguagens imperativas como o C. São enumerados os potenciais benefícios da linguagem e apresenta-se uma panorâmica geral das suas construções e estruturas de dados. O capítulo conclui com uma breve discussão sobre os dois paradigmas de programação em causa: o imperativo e o funcional.

O terceiro capítulo apresenta a aplicação que servirá de base aos testes, a Aplicação de Simplificação e Derivação de Expressões Matemáticas, desenvolvida em Haskell. Descreve-se o objectivo da aplicação, a sua arquitectura e blocos funcionais.

O quarto capítulo efectua um estudo da *framework* HUnit 1.0, que serve de base à arquitectura de testes desenvolvida. São abordados os conceitos de asserções, testes e casos de teste posicionado-os no contexto da *framework*. São aplicadas as suas principais funcionalidades de contagem de casos de teste, identificação de caminhos em hierarquias de testes, execução de testes e outras mais avançadas.

O quinto capítulo descreve os componentes da arquitectura do ambiente de testes. Estes componentes incluem a *framework* HUnit 1.0, a aplicação a testar, a execução de testes e a interface. É também apresentado um dialecto XML de descrição de testes unitários em Haskell, denominado HtestML.

O sexto capítulo aborda todo o processo de concepção de casos de teste. São enunciadas as regras de especificação da aplicação em teste e organizados os casos de testes que as verificam. Esses casos de teste são classificados em três grandes categorias: conversão de formatos entre expressões, simplificação de expressões e cálculo de derivadas.

O sétimo capítulo descreve a execução dos testes e analisa os resultados obtidos. São identificados os casos de teste que acusam falhas na aplicação e discutida a sua gravidade e impacto na aplicação.

Encerra-se o relatório com uma conclusão que discute os objectivos atingidos com o trabalho.

Capítulo 2

Breve introdução à linguagem Haskell

Neste capítulo introduz-se a linguagem em que a arquitectura de testes e o sistema a ser testado foram desenvolvidos – a linguagem Haskell. O objectivo é enquadrar a linguagem no contexto dos paradigmas de programação e descrever as suas principais características. São abordadas também algumas construções e estruturas de dados da linguagem.

2.1 O paradigma funcional

C, Java, Pascal, Ada, entre outras são linguagens imperativas, algumas incluindo orientação por objectos. São "imperativas" no sentido em que consistem numa sequência de comandos que são executados uns após os outros. Um programa funcional pode consistir numa uma expressão simples, que ao ser executado avalia essa expressão.

Um exemplo simples de programação funcional é a utilização de uma folha de cálculo. O valor de uma célula pode ser exprimido em função dos valores de outras células. O foco é sempre naquilo que está a ser calculado e não na forma como deve ser calculado. Não é indicada a ordem pela qual as células devem ser calculadas, essa responsabilidade é deixada à própria folha de cálculo. Também não são indicadas quaisquer técnicas de gestão de memória, esperando-se, pelo contrário, que a folha de cálculo pareça ser infinita. Tudo o que indica é o valor de uma célula através de uma expressão (cujas componentes podem ser avaliadas por qualquer ordem), em vez de uma sequência de comandos que calculem o ser valor. Uma consequência de não haver ordem pré-estabelecida é que a noção de atribuição não é muito útil. Se não se conhece o momento em que uma atribuição ocorre, tal não pode ser incorporado na expressão. É este aspecto que fomenta o maior contraste com linguagens como o C em que a sequência de atribuições tem que ser cuidadosamente planeada. O facto do foco passar do conceito de baixo nível "como" para o de alto nível "o quê" é a característica mais marcante que distingue as linguagens funcionais das imperativas.

Outra linguagem quase-funcional é a SQL para interrogação e manutenção de bases de dados. Uma interrogação SQL é uma expressão que involve projecções, selecções, junções e condições. A expressão indica quais as relações que devem ser interrogadas e o que se pretende, no entanto, não fornece qualquer indicação de como tal deve ser obtido. As implementações de SQL incluem algoritmos de optimização de interrogações que têm como principal objectivo determinar qual a melhor ordem para avaliar a expressão. A execução desses algoritmos é transparente para o programador.

2.2 A linguagem Haskell

Haskell é uma linguagem de programação criada em 1990 e enquadrada no paradigma da programação funcional. Outras exemplos de linguagens funcionais são Lisp, Scheme, Erlang, Clean, Mercury, ML, OCaml, SQL, XSL, etc. É uma linguagem polimórfica, altamente tipada, com modelo de avaliação *lazy* e considerada pura no contexto das linguagens funcionais. O seu nome presta homenagem a Haskell Brooks Curry, cujo trabalho matemático serve de base às linguagens funcionais, nomeadamente no que respeita a *lambda calculus*.

A principal motivação para o aparecimento de linguagens funcionais como o Haskell, é o de tornar o processo de escrita e manutenção de grandes sistemas de software mais simples e económicos. Os criadores da linguagem argumentam como vantagens de utilização o aumento da produtividade dos programadores, a produção de código mais curto, mais legível e de manutenção mais simples. Argumentam também a diminuição de erros e maior fiabilidade e a redução do fosso semântico entre o programador devido ao maior poder expressivo.

A linguagem Haskell tem um espectro de aplicação alargado servindo para uma grande variedade de aplicações. Segundo os seus autores, é particularmente adequada para programas com grande necessidade de modificação e manutenção. A maior parte do ciclo de vida de um produto de software corresponde à especificação, desenho e manutenção e não propriamente à programação. As linguagens funcionais são vocacionadas para escrever especificações que podem ser executadas e consequentemente testadas. Essas especificações constituem o primeiro protótipo do programa final. Os programas nestas linguagens são relativamente simples de manter porque o código é mais curto, mais claro e o controlo rigoroso de efeitos laterais elimina uma grande quantidade de interacções imprevistas.

Um exemplo típico que pretende ilustrar estes conceitos é a implementação do algoritmo de ordenação de um vector pelo método *Quicksort*, apresentado-se a codificação em Haskell e em C.

A implementação em Haskell pretende ser concisa e exprimir a intenção do desenho do método:

```
qsort []     = []
qsort (x:xs) = qsort lt ++ [x] ++ qsort ge
               where
                 lt   = [y | y <- xs, y < x]
                 ge = [y | y <- xs, y >= x]
```

A implementação em C é mais densa e complexa:

```
qsort( a, lo, hi ) int a[], hi, lo; {
  int h, l, p, t;

  if (lo < hi) {
    l = lo; h = hi; p = a[hi];
    do {
      while ((l < h) && (a[l] <= p))
          l = l+1;
      while ((h > l) && (a[h] >= p))
          h = h-1;
      if (l < h) {
          t = a[l]; a[l] = a[h]; a[h] = t;
      }
    } while (l < h);
    t = a[l]; a[l] = a[hi]; a[hi] = t;
    qsort( a, lo, l-1 );
    qsort( a, l+1, hi );
  }
}
```

Este exemplo-limite é frequentemente utilizado pelo entusiastas da linguagem Haskell para ilustrar o seu poder expressivo. Tendo este exemplo como base, são enumerados os seguintes benefícios da linguagem Haskell e da programação funcional em geral:

1. **Concisão**
   Programas escritos em linguagem funcionais tendem a ser mais concisos do que os seus equivalentes em linguagens imperativas. Mesmo atendendo ao facto de que a implementação do algoritmo *quicksort* constitui um caso extremo, de uma forma geral a redução em linhas de código pode variar numa razão de 2 a 10.

2. **Facilidade de compreensão**
   Segundo os seus autores, é possível compreender a implementação do *quicksort* sem qualquer conhecimento prévio de Haskell. A primeira linha afirma que o resultado de ordenar uma lista vazia é uma lista vazia. A segunda linha afirma que para ordenar uma lista cujo primeiro elemento é x e a parte restante é xs, basta ordenar os elementos de xs que são inferiores a x, os elementos de xs que são iguais ou superiores a x e concatenar os resultados com x no meio. Como as sub-listas resultantes são inferiores à lista original o processo acaba por convergir para as sub-listas vazias. A implementação em C não é de fácil compreensão. É necessário seguir o fluxo de execução e as variáveis utilizadas, sendo também mais passível de erros de programação.

3. **Forte sistema de tipos**
   A maioria das linguagens funcionais, e o Haskell em particular, são fortemente tipadas, eliminando uma grande quantidade de erros comuns durante a compilação. De uma forma geral um sistema de tipo permite que o programador indique *à piori* como devem ser tratados os dados, associado-os a um tipo. Com um forte sistema de tipos não existe, por exemplo, a possibilidade de tratar um inteiro como um apontador ou seguir um apontador nulo.

4. **Avaliação *lazy***
   Muitas linguagens funcionais só avaliam as partes do programa que são necessárias para o cálculo que é efectuado. Designa-se a este tipo de avaliação por *lazy*. As estruturas de dados, como as listas, são avaliadas só até onde for necessário podendo haver partes delas que nem são avaliadas.

5. **Abstracção poderosa**
   As funções de ordem elevada são um mecanismo de abstracção que é oferecido pelas linguagens funcionais. Consiste em poder usar funções como argumentos, parâmetros de saída, armazenamento em estruturas de dados e qualquer outra operação que possa ser efectuada com valores de qualquer tipo de dados. A principal vantagem é que a utilização de funções de ordem elevada pode aumentar bastante a estrutura e modularidade de muitos programas.

6. **Gestão automática de memória**
   Muitos programas sofisticados precisam de alocar memória dinamicamente na *heap*. Em C utilizam-se funções específicas (`malloc`) para alocar o espaço e inicializá-lo. O programador é também responsável por libertar a área alocada, caso contrário fica sujeito à ocorrência de apontadores vazios. As linguagens funcionais libertam o programador destas tarefas e à semelhança do Java possuem mecanismos de *garbage collector* que se encarregam de assumir toda a gestão de memória de forma automática.

2.3 Construções e estruturas de dados

Nesta secção abordam-se de forma muito breve as principais construções e estruturas de dados da linguagem Haskell.

As expressões simples incluem literais, operadores e aplicações de funções:

```
0 5 (-1) 3.14159 'c'

7 + 9

abs (-4)
sqrt 4.0
```

Embora seja possível indicar tipos o operador `::`, normalmente deixa-se que o compilador faça inferência de tipos.

Os tipos primitivos são os seguintes:

- `Int` – Inteiro de precisão fixa
- `Integer` – Inteiro de precisão arbitrária
- `Float` – Fraccionário de precisão fixa
- `Double` – Fraccionário de precisão dupla
- `Char` – Caracter Ascii
- `Bool` – Valor booleano, `True` ou `False`

Existem também tipos agregados:

- Tuplos – Sequência de elementos de tipos pré-existentes podendo ser diferentes uns dos outros. O número de elementos está fixado na definição do tipo. Um exemplo é o tipo `(Int, Float, String)`, que pode ser instanciado com o tuplo `(42, 3.14159, "Hello, world!")`.
- Listas – Sequência ordenada de elementos de um tipo, sendo o número de elementos arbitrário. Alguns exemplos são a lista vazia `[]`, uma lista de inteiros `[1, 2, 3]` e uma *string* (lista de caracteres) `"abc"`. Note-se que podem haver listas de tuplos ou mesmo listas de listas.

A linguagem tem construções `let`, como o exemplo seguinte:

```
let
   y = x + 2
   x = 5
in
   x / y
```

A expressão `let` não se destina a permitir atribuições já que `y` e `x` não são variáveis, tratam-se apenas de identificadores para as expressões calculadas, que não podem ser re-utilizados.

A estrutura condicional básica é a *if-then-else*, encontrada nas linguagens imperativas. A principal diferença é que a expressão nos ramos `then` e `else` têm que avaliar para um valor e têm que ser do mesmo tipo:

```
if a == b
then "foo"
else "bar"
```

A definição de uma função pode ser acompanhada de uma assinatura que indica os tipos dos argumentos e do valor retornado. Um exemplo é uma função simples que some dois números:

```
add :: Int -> Int -> Int
add x y = x + y
```

A função `add` tem dois argumentos de tipo `Int` e retorna um valor de tipo `Int`.

A mesma função pode ser definida com um único parâmetro, um tuplo par, que agregue os dois valores a somar:

```
add :: (Int, Int) -> Int
add (x, y) = x + y
```

As funções podem ser assim definidas como um conjunto de equações em que são tratados os vários casos possíveis de parâmetros de entrada. Um exemplo é a função que soma todos os valores de uma lista de inteiros. A função trata separadamente o caso excepcional (lista vazia) e o caso geral. Note-se a utilização da recursividade, muito frequente em programas funcionais. O primeiro elemento da lista é denotado por `x` e a parte restante (cauda da lista) é denotada como a sub-lista `xs`.

```
sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs
```

Uma construção particularmente útil para detalhar diferentes comportamento dentro de uma função são as denominadas *guardas* (denotadas pelo símbolo |), que permitem isolar os casos importantes em que a função altera o seu comportamento. Um exemplo é a seguinte definição da função `abs`:

```
abs :: Integer -> Integer
abs 0 = 0
abs x
  | x > 0 = x
  | x < 0 = -x
```

Existem mais algumas construções para além das apresentadas possibilitando um bom poder expressivo na formulação de algoritmos. É também bastante rica em termos de bibliotecas de funções e *frameworks*, estando continuamente em desenvolvimento novas extensões da linguagem.

A partir do *website* da linguagem, `http://www.haskell.org/`, é possível ter acesso a inúmeros tutoriais introdutórios bem como documentação extremamente aprofundada da linguagem. São disponibilizadas muitas bibliotecas que podem ser livremente importadas. Algumas dessas bibliotecas extendem bastante a linguagem dotando-a da possibilidade de poder rivalizar com muitas linguagens imperativas.

2.4 Paradigma imperativo ou funcional?

A implementação do *quicksort* em C utiliza uma técnica bastante sofisticada, inventada por Hoare, em que o vector é ordenado directamente, sem recorrer a outras estruturas de dados. Em resultado disso, o algoritmo é executado muito rapidamente e exigindo muito pouca memória para além da necessária para conter o vector. A implementação em Haskell por outro lado aloca muito mais memória e é de execução muito mais lenta que o a execução em C. A implementação em C troca a complexidade por tempos de execução reduzidos. Tal leva a concluir que em aplicações em que o desempenho que se baseie em tempos de execução seja um factor critico, as linguagens como C serão sempre uma melhor escolha do que o Haskell, precisamente porque lidam mais como a forma como as computações são efectuadas. A distancia do Haskell ao *hardware* é maior do que a do C, por isso consegue exprimir os algoritmos a um nível mais elevado, mas com menor nível de controlo sobre a máquina.

Verifica-se que são poucos os programas que exigem grande desempenho medidos somente tendo por métrica o tempo de execução. Os entusiastas do Haskell defendem que se abandonou a escrita de programas em Assembler, deixando-a apenas para casos extremos de desempenho. Argumentam que os benefícios de se ter um modelo de programação mais próximo da intenção do programador compensa os custos que tal implica.

Uma visão mais ousada posiciona as linguagens funcionais relativamente às imperativas da mesma forma que estas se posicionam relativamente ao Assembler. Provavelmente trata-se de uma assunção megalómana mas em termos estritos verifica-se que em ambos os casos, os programas têm maior poder expressivo e menor controlo directos sobre a máquina. Da mesma forma verifica-se que para a maioria dos programas, este compromisso é aceitável.

Capítulo 3

Aplicação de simplificação e derivação de expressões matemáticas

3.1 Objectivo

O objectivo da aplicação é solicitar uma expressão ao utilizador e calcular a expressão simplificada e a sua derivada. A aplicação foi desenvolvida em Haskell como caso de estudo de linguagens funcionais.

A sintaxe das expressões matemáticas é próxima da notação algébrica tendo como principais características possibilidade de omissão de operadores implícitos, indicação de precedências de operadores, notação infixa de operadores e inclusão de algumas funções conhecidas.

Os operadores suportados são os da adição, subtracção, multiplicação, divisão, potenciação e radiciação. A aplicação suporta algumas funções trigonométricas, a função logarítmica e a função exponencial. Os valores numéricos das expressões podem ser quaisquer números reais, sendo reconhecida apenas uma variável.

Para garantir a robustez do sistema, a aplicação inclui mecanismos de validação que testam a expressão introduzida do ponto de vista sintáctico e semântico. A validação sintáctica permite detectar situações de erros de sintaxe como a existência de símbolos não permitidos, sequências de operadores não permitidas, funções não suportadas, balanceamento de parêntesis, etc. Se um expressão estiver correcta do ponto de vista sintáctico, é efectuada de seguida uma validação semântica para detectar situações de violações de domínio das funções, divisões por zero ou radicandos inválidos. Só são efectuadas simplificações e derivações de expressões que tenham sido examinadas e aprovadas pelo validador tanto do ponto vista sintáctico como semântico.

As secções seguintes apresentam a gramática das expressões, a arquitectura da aplicação e os verificações efectuadas nas validações.

3.2 Gramática das expressões

A aplicação efectua cálculos com números reais, exprimindo o resultado aproximado com números racionais sob a forma decimal. A notação de variável é explicitada por um x. De forma a tornar mais rápida e eficiente a introdução de expressões, os operadores matemáticos são representados por caracteres específicos. Por exemplo, a multiplicação, identificada na notação matemática pelo símbolo

$\times$ deve ser indicada pelo caracter $*$. A prioridade dos operadores é também a existente na notação matemática normal. A aplicação suporta as seguintes funções reais de variável real: polinómios, funções trigonométricas e trigonométricas inversas, logarítmica e exponencial.

O formato com que o utilizador introduz uma expressão é designado por *formato externo*. Embora na validação e processamento, ocorram transformações da expressão em outros formatos, o resultado das expressões produzidas obedece também a este formato.

Uma definição formal desse formato é obtida através de uma gramática que explicita os operadores e funções admissíveis, bem como as sequências permitidas entre eles. A figura 3.1 apresenta a gramática da aplicação através da sua escrita em EBNF (*Extended Backus-Naur Form*):

$\langle expressao \rangle ::= '('\{\langle expressao \rangle \langle operador \rangle \langle expressao \rangle | \langle funcao \rangle (\langle expressao \rangle) | \langle inteiro \rangle | \langle real \rangle | x\}')'$
$\langle funcao \rangle ::= sen|cos|tg|arcsen|arccos|arctg|log|exp$
$\langle operador \rangle ::= +|-|*|/|\wedge|$
$\langle digito \rangle ::= 0|1|2|3|4|5|6|7|8|9$
$\langle inteiro \rangle ::= \langle digito \rangle | \langle inteiro \rangle \langle digito \rangle$
$\langle real \rangle ::= \langle inteiro \rangle.\langle fraccao \rangle$
$\langle fraccao \rangle ::= \langle digito \rangle | \langle digito \rangle \langle fraccao \rangle$

Figura 3.1 Gramática da aplicação representada em EBNF

As expressões podem ser constituídas por números (inteiros ou racionais), variáveis, operadores, funções e outras sub-expressões.

A tabela 3.1 apresenta alguns exemplos de escrita de expressões matemáticas em notação algébrica e na notação da gramática da aplicação:

Notação algébrica	**Notação da aplicação**
$4x^2 + 2x - 3.05$	4x∧2+2x–3.05
$x(x+2)(x+3)$	x(x+2)(x+3)
$\frac{x+1}{x^2+25x}$	(x+1)/(x∧2+25x)
$\sqrt[3]{(2x+1)^2}$	3|(2x+1)∧2
$sen^2(x) + cos^2(x)$	sen(x)∧2+cos(x)∧2
$3sen(3x+4) \times cos(4-2x)$	3sen(3x+4)∗cos(4-2x)
$(x^2+3)^{sen^4(3x)}$	(x∧2+3)∧(sen(3x)∧4)
e^x	exp(x)
$(\frac{log(x)}{x})^2$	(log(x)/x)∧2

Tabela 3.1 Exemplos de expressões matemáticas

3.3 Arquitectura da aplicação

O utilizador introduz a expressão no formato externo, que é um formato simples e amigável. No entanto, este formato não é o mais conveniente para os processamentos que a aplicação efectua na simplificação e na derivação da expressão introduzida.

Para realizar os cálculos necessários, a aplicação efectua sucessivas transformações da expressão inicial, alterando-lhe o formato, convertendo-a para um formato interno e obtendo os seus elementos simples, formando uma estrutura lógica designada por *representação da expressão em árvore*. Esta representação corresponde à decomposição sintáctica da expressão e é a mais conveniente para efectuar processamentos seguintes.

A estratégia de solução implicou que tenham sido criados blocos funcionais para a execução de tarefas específicas de cada transformação. A execução desses blocos segue uma sequência definida, numa lógica semelhante a uma linha de montagem, que pode ser organizada em três grupos: preparação da expressão, transformação da expressão e apresentação da expressão transformada.

A arquitectura segue um padrão típico em linguagens funcionais, ou seja, a de *pipes and filters*. A figura 3.2 apresenta o diagrama da arquitectura da aplicação com os blocos constituintes de cada grupo.

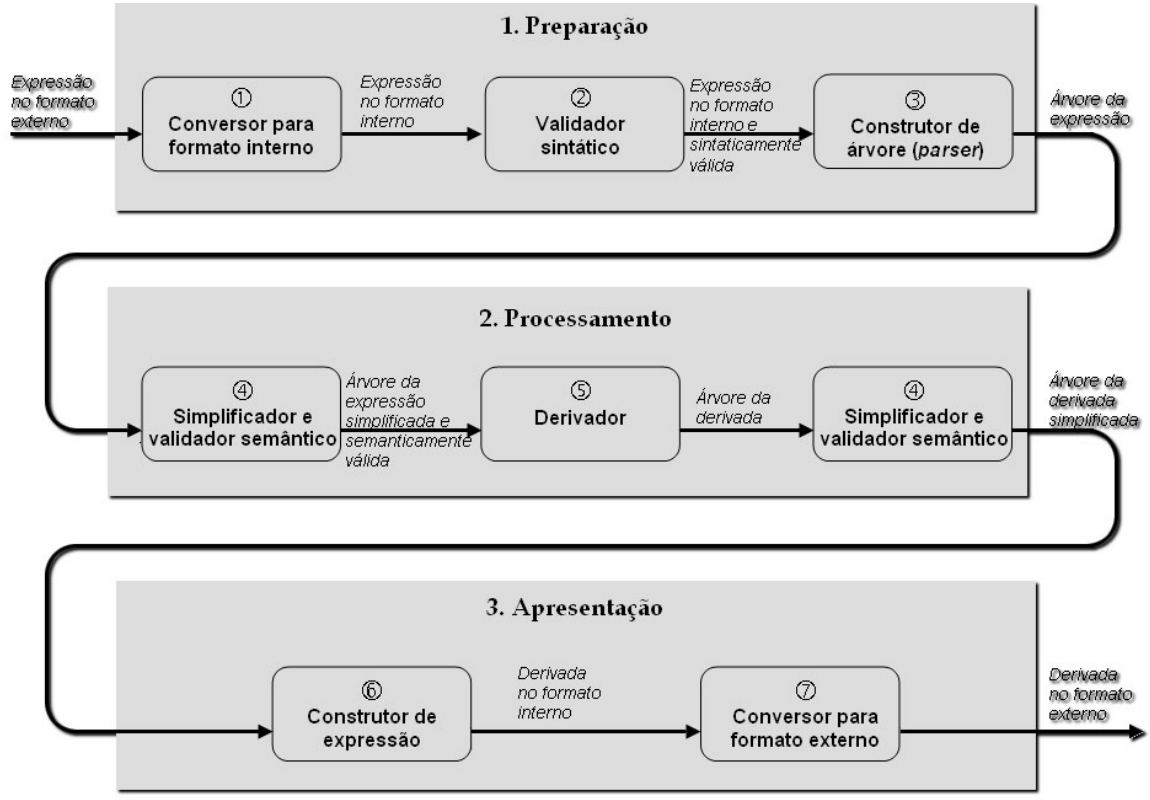

Figura 3.2 Arquitectura da aplicação

Cada bloco funcional da arquitectura corresponde a uma função específica, actuando o bloco como um filtro. Os blocos efectuam processamento e sempre que necessário alteram o formato da entrada em outro que constitui a entrada para o bloco seguinte.

3.4 Blocos funcionais da aplicação

1. **Conversor para o formato interno**
   O formato externo permite que a introdução de expressões seja simples e amigável para o utilizador, possibilitando variantes sintácticas que facilitam essa tarefa (*syntatic sugar*). Por serem permitidas omissões de operações, este formato não se torna conveniente para a análise e decomposição da expressão em elementos. O formato interno é aquele em que todas as variantes sintácticas são substituídas pelas suas representações completas, ocorrendo algumas transformações à expressão. O formato em que a expressão fica após essas transformações designa-se por *formato interno*.

2. **Validador sintáctico**
   Após ser convertida para o formato interno, a expressão pode ser submetida a testes de validação do ponto de vista sintáctico. O objectivo é detectar se a expressão obedece à gramática definida para a aplicação. Se houver alguma violação dessa gramática, a expressão é recusada, não sendo sujeita a mais processamento sendo o utilizador é informado do erro ocorrido. A validação é efectuada através da verificação da ocorrência de determinadas situações. A tabela 6.2 indica as verificações efectuadas e exemplos de situações de ocorrência de erro:

3. **Construtor de árvore (*parser*)**
   Se a expressão, no formato interno, for válida, está em condições de ser substituída pela sua representação em árvore. A árvore (*parse-tree*) é construída através da análise lexical de cada caracter da expressão, sendo o resultado final um conjunto de elementos com uma organização hierárquica. Cada elemento pode ser um número, uma variável, uma operação ou uma função e possui uma estrutura própria. A análise do processo de construção da parse-tree está fora do âmbito deste trabalho, bastando apenas a percepção que a árvore é uma estrutura recursiva do seguintes tipo:

   ```
   Elemento (Tipo, Coeficiente, Expoente, [Elemento])
   ```

   Os tipos podem ser constantes, variáveis, operações (adição, multiplicação e potenciação) e funções (seno, co-seno, tangente, co-tangente, logarítmo e exponencial). Note-se que não são decompostas operações de divisão e radiciação. Esta omissão deve-se ao facto da aplicação, em vez de ser forçada a tratar mais dois casos, identificar uma divisão como uma multiplicação pelo inverso de um número e uma radiciação como uma potência de expoente fraccionário.

4. **Simplificador e validador semântico**
   O simplificador analisa a representação em árvore da expressão e procura reduzir essa árvore, através de um conjunto de operações, simplificando somas, produtos, potências, aplicação de funções entre outras. Paralelamente à tarefa de simplificação, este bloco também é responsável por efectuar a validação semântica, ou seja, verificar se as sub-expressões, após tentativa de simplificação têm significado sob ponto de vista matemático. A tabela **??** indica as verificações efectuadas e exemplos de situações de erros: A partir da representação inicial da árvore da expressão não é possível determinar se existem violações de domínio, uma vez que só durante a

simplificação é que são efectuadas as operações algébricas. Antes de efectuar qualquer operação que possa levar a violação de domínio, os operandos são testados. Se for detectada uma das três situações anteriores, a operação não é efectuada e o erro é colocado na própria árvore da expressão.

5. **Derivador**
   O derivador analisa a representação em árvore da expressão e cria novos elementos para cada elemento analisado. O conjunto dos novos elementos criados constitui a representação em árvore da derivada. A regra de derivação fornece indicações de como construir o elemento da derivada correspondente a uma expressão. São indicadas operações e o papel dos sub-elementos como operandos. É frequente também a necessidade de derivar os sub-elementos e incluir a sua derivada na árvore. Cada elemento da árvore da expressão é substituído por um outro que corresponde à regra de derivação a aplicar a esse tipo. A primeira regra de derivação a aplicar é obtida a partir do primeiro elemento da árvore, que corresponde à operação de menor prioridade da expressão. O processo de derivação, tal como o de simplificação, é recursivo. Uma vez obtida toda a árvore resultante de uma derivação, essa árvore é simplificada pelo mesmo bloco que simplificou inicialmente a expressão.

6. **Construtor de expressão**
   Para ser possível apresentar o resultado final do processamento da representação em árvore de uma expressão, é necessário convertê-la novamente para uma expressão no formato interno (com notação infixa).

7. **Conversor para o formato externo**
   Para se terminar a fase de apresentação do resultado, é apenas necessário omitir o operador de adição nas parcelas de valor negativo. Após esta conversão, a expressão está no formato externo (em linguagem algébrica de fácil compreensão pelo utilizador) e pode ser apresentada.

Capítulo 4

Framework de testes HUnit 1.0

A HUnit é uma *framework* de testes unitários para a linguagem Haskell, inspirada na ferramenta JUnit de igual propósito para Java. Foi desenvolvida em 2002 por Dean Herington encontrando-se ainda na sua versão 1.0.

Utilizando a HUnit, tal como a JUnit, é simples criar testes, atribuir-lhes nomes, agrupá-los em suites e executá-los, deixando a verificação dos resultados ao cargo da *framework*. A especificação de testes em HUnit é ainda mais flexível do que em JUnit, devido à natureza da linguagem Haskell. De momento, a HUnit só inclui um controlador baseado em texto, no entanto, a *framework* está desenhada de forma a ser facilmente extendida e acomodar outros tipos de controladores.

As secções seguintes exploram a *framework* com exemplos da utilização na Aplicação de simplificação e derivação de expressões matemáticas apresentada previamente.

4.1 Asserções

Uma *asserção* é o bloco mais básico para a construção de um teste e define-se como sendo a afirmação de que uma proposição é verdadeira. No contexto de uma linguagem de programação, essa proposição pode assumir a forma de uma condição passível de ser avaliada como verdadeira ou falsa. A asserção sobre essa condição equivale a afirmar que ela é verdadeira.

As asserções são combinadas de forma a serem incluídas em casos de teste e de igual forma, vários casos de teste formam uma suite de testes.

No contexto da linguagem Haskell, a *framework* HUnit define um tipo de dados específico para representar asserções, ou seja, como uma computação IO:

```
type Assertion = IO ()
```

O tipo `IO` indica uma computação da mesma forma com que o tipo *Integer* indica inteiro e *Bool* indica um valor lógico. Numa abordagem simples, pode pensar-se em computações como funções que aceitam um estado do sistema como argumento e retornam um par constituído pelo estado do sistema actualizado e um resultado. Dentro dessa função ocorre uma sequência de computações de tipo `IO` que podem alterar o estado do sistema. Exemplos disso serão a entrada e saída de caracteres ou a leitura e escrita de ficheiros.

Num programa em Haskell, as expressões do tipo `IO` comportam-se como quaisquer outras expressões. Na realidade, o tipo `IO` insere-se numa classe especial de construções da linguagem Haskell denominadas Monads, que é um dos seus tópicos mais complexos. A discussão de Monads está fora do âmbito desta análise e não é crítica para os seus objectivos. É suficiente encarar o tipo `IO` como um tipo que retorna informação para o exterior.

As asserções são assim computações `IO` que embora possam alterar o estado do sistema ao produzirem resultados para o exterior, não retornam resultados como as funções tipicamente fazem. A motivação das asserções serem representadas como computações `IO` é para permitir que programas com efeitos laterais possam ser testados. A utilidade de uma asserção é a sinalização de um sinal de erro, invocando a função `assertFailure`, que lança uma excepção assinalada por uma mensagem:

```
assertFailure :: String -> Assertion
assertFailure msg = ioError(userError ("HUnit:" ++ msg))
```

Na realidade, a função `assertFailure` é ela própria uma computação já que na sua assinatura, o valor retornado é de tipo `Assertion`, logo de tipo `IO`.

A *framework* define três tipos de asserções. Para cada tipo, a função de construção da asserção é diferente, no entanto, o resultado é sempre de tipo `Assertion`. Todas utilizam a computação `assertFailure` quando pretendem sinalizar que a asserção é falsa.

Utilizando o mecanismo de *binding* (que não é uma atribuição, já que tal construção não existe em linguagens funcionais), é possível associar um identificador a uma asserção. Os tipos de asserções são os seguintes:

- **Booleana**
  Afirma que uma expressão booleana é verdadeira. Pode ser indicada uma mensagem de erro para o caso em que a asserção seja avaliada como falsa quando executada num caso de teste.

  ```
  assertBool :: String -> Bool -> Assertion
  assertBool msg b = unless b (assertFailure msg)
  ```

  A aplicação de simplificação e derivação de expressões matemáticas inclui uma função `isNumber` que verifica se uma *string* pode ser tratada como um número, ou seja, se não contem caracteres não numéricos. A função retorna um valor booleano, pelo é adequada para ser testada por este tipo de asserção. Com esta função é possível definir duas asserções booleanas:

  ```
  assercao1 = assertBool "isNumber 2.5" (isNumber "2.5")
  assercao2 = assertBool "isNumber 2a5" (isNumber "2a5")
  ```

  Quando forem incluídas num caso de teste, a primeira asserção seria avaliada como verdadeira e a segunda como falsa pois inclui o caracter `a` que é não numérico.

- ***String***
  Afirma que uma *string* é nula. A mensagem de erro é a própria *string*, caso a asserção seja avaliada como falsa quando executada num caso de teste.

```
assertString :: String -> Assertion
assertString str = unless (null str) (assertFailure str)
```

A aplicação de simplificação e derivação de expressões matemáticas inclui uma função `substr` que extrai uma *substring* de outra mediante a indicação de posições de inicio e fim. A função retorna uma *string* resultante da extracção, pelo que é adequada pode ser testada por este tipo de asserção. Com esta função é possível definir duas asserções de *string*:

```
assercao1 = assertString (substr "2x+3" 0 0)
assercao2 = assertString (substr "2x+3" 1 3)
```

Quando forem incluídas num caso de teste, a primeira asserção seria avaliada como verdadeira, retorna uma *string* nula, e a segunda como falsa pois retorna uma *string* não nula.

- **Igualdade**
  Afirma que duas expressões são iguais. As expressões podem ser de qualquer tipo, no entanto, têm que entre elas ser do mesmo tipo. Pode ser indicada uma mensagem de erro para o caso em que a asserção seja avaliada como falsa quando executada num caso de teste.

```
assertEqual :: (Eq a, Show a) => String -> a -> a -> Assertion
assertEqual mensagem esperado avaliado =
    unless (avaliado == esperado) (assertFailure mensagem)
        where msg = (if null mensagem
                     then ""
                     else mensagem ++ "\n") ++ "expected: " ++
                          show esperado ++ "\n but got: " ++ show avaliado
```

A aplicação de simplificação e derivação de expressões matemáticas inclui uma função `simplif` que simplifica uma expressão matemática indicada numa string. A função retorna uma *string* resultante da simplificação, pelo que é adequada pode ser testada por este tipo de asserção. Com esta função é possível definir duas asserções de igualdade. Além da mensagem de erro, indica-se a expressão esperada e de seguida a expressão que deve ser avaliada para ser comparada com a esperada:

```
assercao1 = assertEqual "Simplificação: 2x+1+3x+4" "5x+5" (simplif "2x+1+3x+4")
assercao2 = assertEqual "Simplificação: 2x+1+3x+4" "8x-2" (simplif "2x+1+3x+4")
```

Quando forem incluídas num caso de teste, a primeira asserção seria avaliada como verdadeira, a expressão esperada é igual à avaliada, e a segunda como falsa, onde tal não acontece.

A definição de asserções por si só não tem aplicação prática, é necessário incluí-las em casos de teste. É durante a execução de casos de teste que se avaliam as asserções de qualquer um dos três tipos.

4.2 Casos de teste

Um *caso de teste* consiste num conjunto de uma ou várias asserções. Os casos de teste são independentes entre si, ou seja, a execução de uns não está dependente da execução de outros. Por outro lado, se um caso de teste incluir várias asserções, a sua execução não é independente. Se uma asserção no conjunto for avaliada como falsa, as seguintes já não serão avaliadas. Este raciocínio é útil para definir uma lógica de testes incrementais em que existem funcionalidades que dependem umas das outras. Se uma funcionalidade falhar numa asserção, não fará sentido testar outra que depende do resultado errada da primeira.

Independentemente do número de asserções que possa incluir, a criação de um caso de teste é efectuada com o construtor `TestCase`. O caso de teste pode indicar directamente uma asserção ou referenciar um identificador de uma (ou várias) já criadas. A cada caso de teste pode ser associado um identificador com o mecanismo de binding (como acontece com as asserções).

```
TestCase :: Assertion -> Test

cteste1 = TestCase assercao1
cteste2 = TestCase (assertString (substr "2x+3" 1 3))
```

No primeiro exemplo é criado um caso de teste a partir do identificador `assercao1` e associa-o ao identificador `teste1`. No segundo exemplo é criado um caso de teste a partir de uma asserção de *string* e associa-o ao identificador `teste2`.

No exemplo seguinte, o caso de teste é composto por duas asserções com ordem de precedência:

```
cteste = TestCase (do assertEqual ("fmt_int: "++expressao) esperado fmt_int
                      assertEqual ("fmt_ext: "++expressao) expressao fmt_ext)
  where expressao = "3x^2-3x+4"
        esperado = "3*x^2+-3*x+4"
        fmt_int = ext2int expressao
        fmt_ext = int2ext fmt_int
```

A construção com `do` é necessária para conseguir colocar duas ou mais asserções num caso de teste. A construção `where` define *bindings* a serem utilizados nas asserções.

O objectivo deste caso de teste é verificar se na aplicação de simplificação e derivação de expressões matemáticas uma expressão é correctamente convertida do formato externo para o interno e vice-versa. Sendo funções simétricas, podem ser incorporadas no mesmo caso de teste, mas só faz sentido avaliar a segunda se a primeira produzir resultados satisfatórios.

A expressão associada ao identificador `expressao` é o objecto do teste e encontra-se no formato externo. A expressão associada ao identificador `esperada` corresponde à transformação da expressão no formato interno. É avaliado o formato interno com a função ext2int e associado ao identificador `fmt_int`. A primeira asserção utiliza é construída com base nestes identificadores.

É avaliado o formato externo com a função `int2ext` e associado ao identificador `fmt_ext`. A expressão esperada na segunda asserção é a que é inicialmente convertida para o formato interno, ou seja, associada ao identificador `expressao`.

Garante-se que na execução deste caso de teste, a segunda asserção só será avaliada se a primeira for verdadeira.

4.3 Agrupamento de testes

Define-se um *teste* como um conjunto de casos de teste. A distinção entre teste e caso de teste não existe em HUnit. A função `TestCase` retorna sempre um valor de tipo `Test`. É este facto que permite que um teste possa ser um conjunto de casos de teste ou um conjunto de testes.

Num ambiente de testes, assim que o número de testes começa a aumentar é recomendável formar grupos de testes para os processar mais facilmente. Os identificadores por *binding* facilitam apenas a tarefa do programador de testes. A capacidade de poder atribuir nomes a testes sob a forma de labels (*strings*) facilita a sua identificação em tempo de execução. O agrupamento de testes e a atribuição de nomes a testes (ou grupos de testes) são as duas formas que o HUnit disponibiliza para gerir grandes colecções de testes.

O tipo de dados `Test` é definido de forma a poder construir testes com a máxima flexibilidade possível:

```
data Test = TestCase Assertion
          | TestList [Test]
          | TestLabel String Test
```

Desta definição salientam-se os seguintes aspectos:

- Uma `TestList` é uma lista de testes em vez de uma lista de casos de teste. Tal implica que a estrutura de um teste é uma árvore. Desta forma, os testes podem ser organizados da mesma forma que ficheiros são organizados num sistema operativo.
- O nome, `TestLabel`, é uma *string* associada a um teste em vez de a um caso de teste. Tal implica que a todos os nós da árvore pode ser associado um nome. Esta característica facilita a organização dos testes.
- Uma `TestLabel` é um conceito separado de um caso de teste ou uma lista de testes. Tal implica que é opcional associar um nome a um teste. É benéfico que assim seja, pois em certas situações pode ser desnecessário e inconveniente ter se associar sempre um nome a cada nó de uma grande árvore de testes.

Um exemplo na aplicação de simplificação e derivação de expressões matemáticas é a seguinte suite de testes:

```
tSNumeros = TestCase (assertEqual "Simplificacao:" "9" (simplif "2+7"))
tSMonomios = TestCase (assertEqual "Simplificacao:" "9x" (simplif "2x+7x"))
tSPolinomio = TestCase (assertEqual "Simplificacao:"
                                    "12x^2+4x+5" (simplif "3x^2+9x^2+3x+5+x"))

tSomas = TestLabel "Somas" (TestList [TestLabel "Numeros" tSNumeros,
                                      TestLabel "Monomios" tSMonomios,
                                      TestLabel "Polinomio" tSPolinomio])

tPNumeros  = TestCase (assertEqual "Simplificacao:" "14" (simplif "2*7"))
tPMonomios  = TestCase (assertEqual "Simplificacao:" "14x^2" (simplif "2x*7x"))
```

```
tPPolinomio  = TestCase (assertEqual "Simplificacao:"
                                     "405x^6" (simplif "3x^2*9x^2*3x*5*x"))

tProd  = TestLabel "Produtos" (TestList [TestLabel "Numeros" tPNumeros,
                                         TestLabel "Monomios" tPMonomios,
                                         TestLabel "Polinomio" tPPolinomio])

suiteTP = TestLabel "Polinómios" (TestList [tSomas, tProd])
```

Esta suite de testes agrega testes de simplificação de produtos e somas. A simplificação de somas agrega por sua vez três testes, um com números, outro com monómios e outro com um polinómio. A simplificação de produtos agrega testes idênticos para produtos. Todos os testes têm um nome associado. O teste que define a suite de testes foi nomeado de "Polinomios". Os dois testes agrupadores têm o nome de "Somas" e "Produtos". Os nomes dos testes que agregam outros testes servem para contextualizar os testes elementares. Há um par de testes o nome "Numeros", "Monomios" e "Polinomio" respectivamente. Durante a execução dos testes, como são incluídos os nomes dos testes agrupadores nunca ocorre perda de contexto. Os nomes associados à asserção referem-se à funcionalidade que está a ser testada, que é sempre a simplificação de expressões matemáticas.

A suite de testes pode ser representada por uma árvore de testes, como a da figura 4.1:

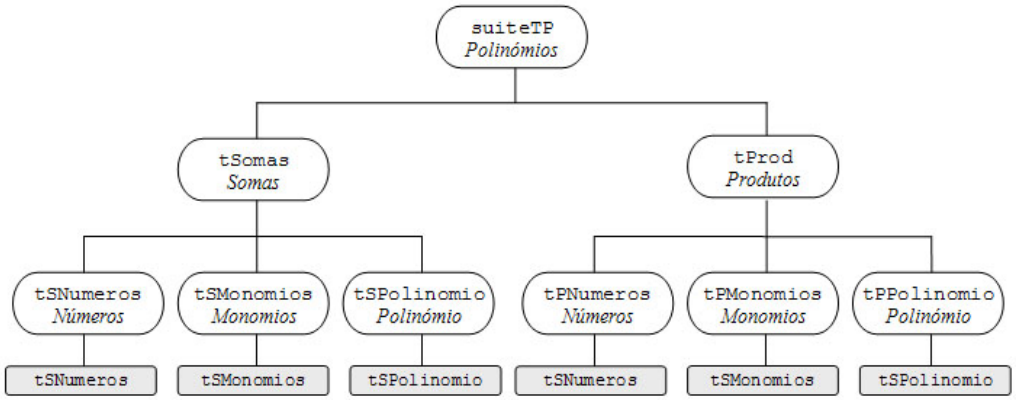

Figura 4.1 Suite de testes de simplificação

4.4 Contagem de casos de teste

É possível contar o número de casos de teste que estão incluídos num nó da árvore, utilizando a função `testCaseCount`.

```
testCaseCount :: Test -> Int
```

A função tem como argumento um teste e retorna um número inteiro que indica quantos testes estão incluídos. O teste indicado como argumento normalmente será um teste agregador já que a contagem

para testes com asserções será sempre de 1 (não incluem outros testes). O resultado da contagem, no entanto, só inclui os testes com asserções, não entrando para a contagem todos os testes que sirvam para agregar outros.

A tabela 4.1 inclui os três casos relevantes de contagem para a suite de testes da figura 4.1:

Comando	Contagem
`testCaseCount tSNumeros`	1
`testCaseCount tSomas`	3
`testCaseCount suiteTP`	6

Tabela 4.1 Contagem de casos de teste da suite de testes

Criando um novo teste na árvore no mesmo nível dos testes agregadores "Soma" e "Produto" para o tratamento de simplificação de potências:

```
tPoten = TestCase (assertEqual "Simplificacao:" "8" (simplif "2^3"))
```

A suite de testes têm que ser alterada de forma a incluir este teste:

```
suiteTP = TestLabel "Polinómios" (TestList [tSomas, tProd, tPoten])
```

Passando a ser representada pela árvore de testes da figura 4.2:

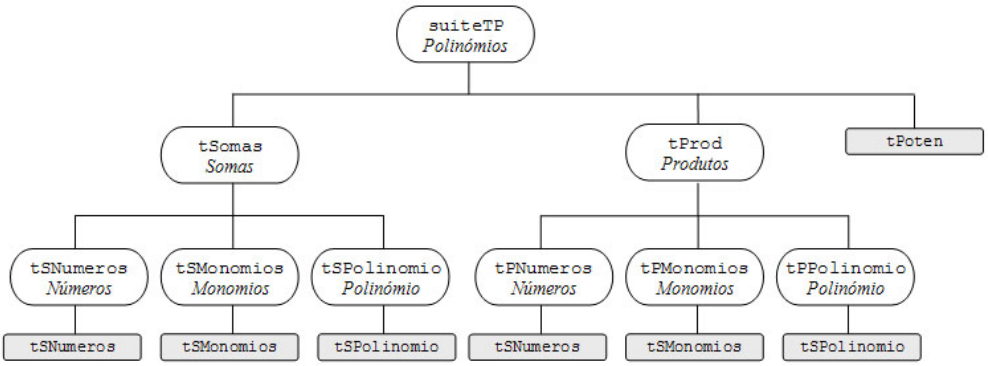

Figura 4.2 Variante à suite de testes de simplificação

A tabela 4.2 inclui os três casos relevantes de contagem para a suite de testes da figura 4.2:

Comando	Contagem
`testCaseCount tSNumeros`	1
`testCaseCount tSomas`	3
`testCaseCount suiteTP`	7

Tabela 4.2 Contagem de casos de teste da variante da suite de testes

O número de casos de teste contado a partir da raiz da arvore aumenta para 7, porque o teste `tPoten` é um teste com uma asserção.

4.5 Caminhos na hierarquia de testes

A forma de identificar univocamente um teste na árvore de testes é indicando o seu caminho desde a raiz. O caminho é definido como a sequência de nós entre a raiz e o teste em causa.

Para o efeito, o HUnit define o tipo de dados `Node` que pode ser um `ListItem` ou uma `Label` e o tipo de dados `Path` que é uma lista de nós:

```
data Node  = ListItem Int | Label String
      deriving (Eq, Show, Read)

type Path = [Node]
```

O tipo `Node` está associado ao tipo `Test`, que define a forma de agrupar testes. Revisitando este tipo:

```
data Test = TestCase Assertion
          | TestList [Test]
          | TestLabel String Test
```

verifica-se que cada sub-lista de testes (`TestList`) é incluída no caminho como uma `ListItem` e cada definição de nome (`TestLabel`) é incluída no caminho como uma `Label`.

A unicidade de um teste é garantida somente pelo inteiro em ListItem, no entanto, se existir um nome para o teste, este é incluído pela `Label`. A numeração dos testes inicia-se com o número zero, seguindo-se o um, dois, etc.

A ordem dos nós no caminho é de baixo para cima, ou seja, do teste para o nó de raiz que agrega todos os testes. Desta forma é mais simples de comparar caminhos já que os que prefixos comuns podem ser partilhados.

Para apresentação dos caminhos que um teste comporta, utiliza-se a função `testCasePaths`. Os caminhos são listados na mesma ordem com que os testes serão executados.

```
testCasePaths :: Test -> [Path]
```

A função recebe um teste, ou seja, qualquer nó da árvore, e apresenta uma lista com todos os caminhos dos testes que esse nó inclui.

Alguns exemplos dos resultados da função aplicados à árvore da figura 4.2:

1. Caminhos a partir de um teste com uma asserção

   ```
   testCasePaths tSNumeros
   [[]]
   ```

   O resultado é uma lista vazia, já que testes com asserções não podem incluir outros testes.

2. Caminhos a partir de um teste agregador de testes com asserções:

   ```
   testCasePaths tSomas
   [[Label "Números",   ListItem 0, Label "Somas"],
    [Label "Monomios",  ListItem 1, Label "Somas"],
    [Label "Polinomio", ListItem 2, Label "Somas"]]
   ```

   O resultado é uma lista com os três testes com asserções com indicação do seu nome (`Label`) e

número (`ListItem`). No fim de cada caminho é também incluído o nome do teste agregador, neste caso *Somas*.

3. Caminhos a partir da raiz da árvore de testes:

```
testCasePaths suiteTP
[[Label "Números",   ListItem 0, Label "Somas",    ListItem 0, Label "Polinómios"],
 [Label "Monomios",  ListItem 1, Label "Somas",    ListItem 0, Label "Polinómios"],
 [Label "Polinomio", ListItem 2, Label "Somas",    ListItem 0, Label "Polinómios"],
 [Label "Números",   ListItem 0, Label "Produtos", ListItem 1, Label "Polinómios"],
 [Label "Monomios",  ListItem 1, Label "Produtos", ListItem 1, Label "Polinómios"],
 [Label "Polinomio", ListItem 2, Label "Produtos", ListItem 1, Label "Polinómios"],
 [ListItem 2, Label "Polinómios"]]
```

   O resultado é uma lista com os todos os caminhos para os casos de teste com asserções, que como se verificou na tabela 4.2, são sete. Note-se que o último caso não tem `Label`, tendo só `ListItem`. Tal deve-se ao facto de não ter sido definida qualquer nome para este teste. A única forma do caminho se referir a ele é indicando que é o teste nº 2 na sequência da árvore.

O resultado dos caminhos da raiz da árvore identifica a framework atribuí automaticamente um número aos testes, como se pode verificar na figura 4.3:

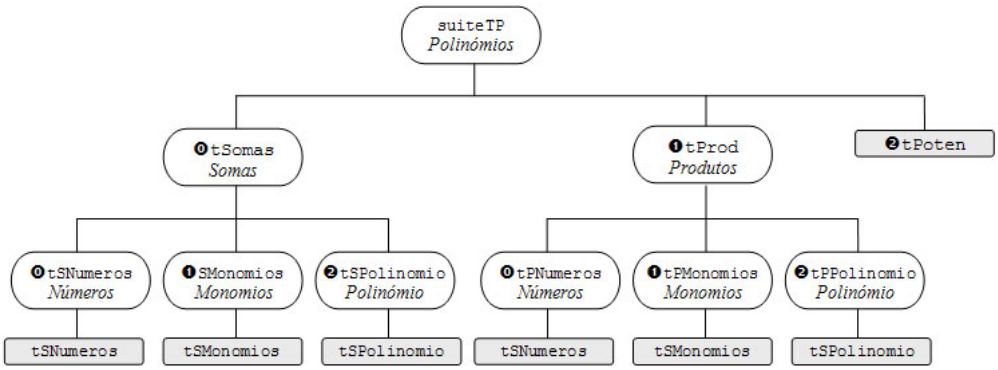

Figura 4.3 Suite de testes de simplificação com numeração

Partindo do teste em direcção à raiz da árvore, a função identificou os caminhos de teste indicados na tabela 4.3, através de números e através de nomes:

Números	**Nomes**
0 → 0	Números → Somas → Polinómios
0 → 1	Monómios → Somas → Polinómios
0 → 2	Polinómio → Somas → Polinómios
1 → 0	Números → Produtos → Polinómios
1 → 1	Monómios → Produtos → Polinómios
1 → 2	Polinómio → Produtos → Polinómios
2	Polinómios

Tabela 4.3 Caminhos na hierarquia de testes

4.6 Execução de testes

Contadores de testes

A execução de um teste envolve um execução série (através do Monad IO) dos seus casos de teste constituintes. Os casos de testes são executados por ordem decrescente de profundidade, ou seja, são executados os mais profundos primeiro e da esquerda para a direita na mesma profundidade. Durante a execução, são mantidos quatro contadores, numa estrutura de dados de tipo `Count`:

```
data Counts = Counts {cases, tried, errors, failures :: Int}
     deriving (Eq, Show, Read)
```

Os quatro elementos da estrutura são valores inteiros cujo valor vai sendo alterado durante a execução de um teste:

- `cases` – Número de testes incluídos no teste, só sendo superior a um para o caso dos testes agregadores. Este valor é uma propriedade estática de um teste e permanece inalterado durante a execução.
- `tried` – Número de testes tentados, ou seja, executados até ao momento (durante uma execução de testes).
- `errors` – Número de testes cuja execução abortou devido à ocorrência de uma excepção inesperada. Estes erros acusam problemas com o teste e não com o código a ser testado.
- `failures` – Número de testes em que pelo menos uma asserção foi avaliada como falsa. Estes erros acusam problemas com o código a ser testado.

Não existe nenhum contador para os casos de teste bem sucedidos. Há dois motivos para que assim seja. Por um lado esse valor é facilmente calculável através de:

```
tried - (errors + failures))
```

O outro motivo é de ordem psicológica. Espera-se sempre que o foco da atenção do testador seja para os casos que falham. Os casos que passam no teste serão tipicamente a maioria. É para a minoria dos casos que falham nos testes que toda atenção deve ser canalizada, daí que o destaque nos próprios contadores seja para essas situações.

Controladores de testes

Os testes em HUnit são executados por um processo denominado Controlador de teste. Este processo tem sempre o mesmo modelo de execução, no entanto, pode apresentar os resultados de forma distinta.

Á medida que a execução do teste progride, são lançados três tipos de eventos para o controlador de teste. Cada controlador pode responder de forma distinta aos eventos, variando normalmente na forma como os apresenta:

- Início – antes da inicialização da execução de teste reporta-se o seu caminho e estado;

- Erro – sempre que um caso de teste termina devido à detecção de um erro, é reportada a mensagem de erro, o caminho e o estado actual do teste;
- Falha – sempre que um caso de teste termina devido a uma falha, a mensagem de falha é reportada, o caminho e o estado actual do teste.

Tipicamente, um controlador de teste apresenta os relatórios de erros e falha imediatamente mas utiliza o relatório de inicio meramente para actualizar o progresso no estado de execução.

O Hunit inclui apenas um controlador de testes que é baseado em texto, no entanto, o utilizador pode definir outros para obter variações na apresentação dos testes. A função que implementa o controlador de testes nativo da framework é a `runTestText`:

```
runTestText :: PutText st -> Test -> IO (Counts, st)
```

O controlador de testes pode ser um pouco adaptado através da indicação de um esquema de apresentação (*reporting scheme*), que corresponde ao primeiro argumento. O teste a executar é indicado no segundo argumento.

Durante a execução, o controlador cria uma *string* para cada evento a reportar e processa-a de acordo com o esquema de apresentação. Quando a execução do teste está completa, o controlador retorna os contadores e o estado final para o esquema de reporte (de tipo par `IO (Counts, st)`)

As *strings* para cada um dos eventos de reporte são as seguintes:

- Um *reporte de início* é o resultado da função `showCounts` aplicado aos contadores imediatamente antes da inicialização do caso de teste a ser executado: A função `showCounts` apresenta o conjunto de contadores da estrutura `Counts` como uma *string*:

  ```
  showCounts :: Counts -> String
  ```

  O resultado tem o seguinte aspecto:

  ```
  "Cases: num_cases  Tried: num_tried  Errors: errors  Failures: failures"
  ```

  onde `num_cases`, `num_tried`, `num_errors` e `num_failures` são os valores contados durante a execução.

- Um *reporte de erro* é apresentado como:

  ```
  "### Error in:   path
   message"
  ```

  onde `path` é o caminho do caso de teste em que ocorreu o erro e `message` é a mensagem que descreve o erro. Se o caminho for vazio, o reporte é apresentado da forma:

  ```
  "### Error:
   message"
  ```

  A função `showPath` é a responsável por apresentar o caminho de um teste num *string*, aceitando um caminho como argumento:

  ```
  showPath :: Path -> String
  ```

  Recorde-se que função que obtém o caminho de um teste (`testCasePaths`) faculta-o com os nós desde o teste até à raiz da árvore. A função `showPath` pelo contrário apresenta o caminho com os nós desde a raiz até ao teste (efectua uma inversão da lista de nós que recebe). Os nós são apresentados separados pelo delimitador `":"`. A representação de `(ListItem n)` é com `(show n)`.

- Um *reporte de falha* é apresentado como:

```
"### Failure in: path
 message"
```

  onde `path` é o caminho do caso de teste em que ocorreu o erro e `message` é a mensagem que descreve o erro. Se o caminho for vazio, o reporte é apresentado da forma:

```
"### Failure:
 message"
```

Esquemas de apresentação

O HUnit inclui dois esquemas de apresentação para controladores baseados em texto. Podem, no entanto, ser definidos outros conforme for a conveniência do programador. Cada esquema baseia-se numa função específica:

1. A função `putTextToHandle` escreve reportes de erro, reportes de falha e o de contagem de final para um *handle* indicado. O *handle* pode ser um ficheiro, um terminal ou qualquer outro dispositivo de saída.

   ```
   putTextToHandle :: Handle -> Bool -> PutText Int
   ```

   Cada um destes reportes é terminado por um carácter de *newline*. Para além disso, se a flag indicada for `True`, também é incluída a escrita do reporte de inicio, não sendo este terminado por uma *newline*. Antes do reporte seguinte ser escrito, o reporte de inicio é "apagado" com uma sequência apropriada de *carriage return* e caracteres de espaço, para produzir os efeitos desejados em terminais. Não é conveniente que assim seja em ficheiros pois produz linhas em branco não necessárias.

2. A função `putTextToShowS` ignora reportes de início e simplesmente acumula reportes e erro e falhas, terminando com *newlines*.

   ```
   putTextToShowS :: PutText ShowS
   ```

   Os reportes de acumulação são retornados (sendo o segundo elemento do par retornado por `runTestText`) como uma função `ShowS` (ou seja, uma com a assinatura `String -> String`) cujo primeiro argumento é uma *string* a ser concatenada às linhas de reporte já acumuladas.

O HUnit fornece a função `runTestTT` que serve de atalho para a utilização mais comum de um controlador baseado em texto. A função `runTestTT` invoca a função `runTestText`, especificando `putTextToHandle stderr True` para o esquema de apresentação (com *handle* para o terminal) e retorna a contagem final da execução do teste.

A implementação é a seguinte:

```
runTestTT :: Test -> IO Counts
runTestTT t = do (counts, 0) <- runTestText (putTextToHandle stderr True) t
                 return counts
```

Exemplos de execução de testes

Exemplos de utilização da função `runTestTT` para execução de testes na árvore da figura 4.2:

- **Execução de um caso de teste isolado**
  Recorde-se a construção do caso de teste de somas num polinómio (corresponde a um dos nós mais profundos da árvore):

```
tSPolinomio = TestCase (assertEqual "Simplificacao:"
                        "12x^2+4x+5" (simplif "3x^2+9x^2+3x+5+x"))
```

  A asserção é avaliada como verdadeira logo a execução não reporta erros, sendo apenas apresentados os reportes de início e de fim:

```
runTestTT tSPolinomio
Cases: 1  Tried: 0  Errors: 0  Failures: 0          (reporte de início)
Cases: 1  Tried: 1  Errors: 0  Failures: 0          (reporte de fim)
```

  Alterando o teste de modo que a asserção seja avaliada como falsa, por alteração do polinómio esperado:

```
tSPolinomio = TestCase (assertEqual "Simplificacao:"
                        "10x^2+8x+3" (simplif "3x^2+9x^2+3x+5+x"))
```

  A execução acusa a presença da falha e contabiliza-a no reporte de fim:

```
runTestTT tSPolinomio
Cases: 1  Tried: 0  Errors: 0  Failures: 0          (reporte de início)
### Failure:                                        (reporte de falha)
Simplificacao:
    expected: "10x^2+8x+3"
    but got: "12x^2+4x+5"
Cases: 1  Tried: 1  Errors: 0  Failures: 1          (reporte de fim)
```

- **Execução de um teste agregador**
  Retirando a falha no caso de teste anterior referente a somas num polinómio, executa-se o teste agregador com o nome *Somas*:

```
runTestTT tSomas
Cases: 3  Tried: 0  Errors: 0  Failures: 0          (reporte de início)
Cases: 3  Tried: 1  Errors: 0  Failures: 0          (reporte de fim do teste 1)
Cases: 3  Tried: 2  Errors: 0  Failures: 0          (reporte de fim do teste 2)
Cases: 3  Tried: 3  Errors: 0  Failures: 0          (reporte de fim do teste 3)
```

  Voltando a "injectar" a mesma falha na soma de polinómios, a execução acusa o erro no caso de teste respectivo na sequência dos outros casos de teste:

```
runTestTT tSomas
Cases: 3  Tried: 0  Errors: 0  Failures: 0          (reporte de início)
```

```
Cases: 3  Tried: 1  Errors: 0  Failures: 0          (reporte de fim do teste 1)
Cases: 3  Tried: 2  Errors: 0  Failures: 0          (reporte de fim do teste 2)
### Failure: Somas:2:Polinomio                      (reporte de falha do teste 3)
    Simplificacao:
    expected: "10x^2+8x+3"
    but got: "12x^2+4x+5"
Cases: 3  Tried: 3  Errors: 0  Failures: 0          (reporte de fim do teste 3)
```

- **Execução da suite de testes**

  Retirando novamente a falha no caso de teste referente a somas num polinómio, executa-se toda a suite de testes com o nome *Polinómios*:

```
runTestTT suiteTP
Cases: 3  Tried: 0  Errors: 0  Failures: 0          (reporte de início)
Cases: 3  Tried: 1  Errors: 0  Failures: 0          (reporte de fim do teste 1)
Cases: 3  Tried: 2  Errors: 0  Failures: 0          (reporte de fim do teste 2)
Cases: 3  Tried: 3  Errors: 0  Failures: 0          (reporte de fim do teste 3)
Cases: 3  Tried: 4  Errors: 0  Failures: 0          (reporte de fim do teste 4)
Cases: 3  Tried: 5  Errors: 0  Failures: 0          (reporte de fim do teste 5)
Cases: 3  Tried: 6  Errors: 0  Failures: 0          (reporte de fim do teste 6)
Cases: 3  Tried: 7  Errors: 0  Failures: 0          (reporte de fim do teste 7)
```

  Injectando a falha anterior e uma outra por alteração do caso de teste:

```
tPMonomios = TestCase (assertEqual "Simplificacao:"
                                   "15x^3" (simplif "2x*7x"))
```

  A execução reporta as duas falhas pela ordem em que os respectivos testes são executados na árvore (note-se a indicação dos caminhos dos casos de teste):

```
runTestTT suiteTP
Cases: 7  Tried: 0  Errors: 0  Failures: 0          (reporte de início)
Cases: 7  Tried: 1  Errors: 0  Failures: 0          (reporte de fim do teste 1)
Cases: 7  Tried: 2  Errors: 0  Failures: 0          (reporte de fim do teste 2)
### Failure in: Polinómios:0:Somas:2:Polinomio      (reporte de falha do teste 3)
Simplificacao:
expected: "10x^2+8x+3"
 but got: "12x^2+4x+5"
Cases: 7  Tried: 3  Errors: 0  Failures: 1          (reporte de fim do teste 3)
Cases: 7  Tried: 4  Errors: 0  Failures: 1          (reporte de fim do teste 4)
### Failure in: Polinómios:1:Produtos:1:Monomios    (reporte de falha do teste 5)
Simplificacao:
expected: "15x^3"
 but got: "14x^2"
Cases: 7  Tried: 5  Errors: 0  Failures: 2          (reporte de fim do teste 5)
Cases: 7  Tried: 6  Errors: 0  Failures: 2          (reporte de fim do teste 6)
Cases: 7  Tried: 7  Errors: 0  Failures: 2          (reporte de fim do teste 7)
```

4.7 Funcionalidades avançadas

O HUnit inclui funcionalidades adicionais para especificar asserções e testes de forma ainda mais concisa e prática. Para a sua implementação, a *framework* utiliza as classes de tipos do Haskell.

Operadores de construções de asserções

Podem ser utilizados os operadores seguintes para construir asserções:

Operador	Asserção criada
`pred @? msg`	`assertionPredicate pred >>= assertBool msg`
`expected @=? actual`	`assertEqual "" expected actual`
`actual @?= expected`	`assertEqual "" expected actual`

Tabela 4.4 Operadores de construção de asserções

No operador `@?` indica-se uma condição booleana e uma mensagem de erro separadamente, tal como na função `assertBool` mas com diferente ordem.

Os operadores `@=?` e `@?=` fornecem uma forma abreviada de criar asserções de igualdade com a função `assertEqual`, quando não é necessária mensagem de erro. Diferem apenas na ordem com que os operandos são indicados (valor esperado e valor avaliado).

No âmbito da aplicação de derivação e simplificação de expressões matemáticas, algumas asserções podem ser escritas mais facilmente:

Operação	Asserção criada
`(isNumber "2.5") @? "isNumber 2.5"`	`assertBool "isNumber 2.5" (isNumber "2.5")`
`"5x+5" @=? (simplif "2x+1+3x+4")`	`assertEqual "" "5x+5" (simplif "2x+1+3x+4")`
`(simplif "2x+1+3x+4") @?= "5x+5"`	`assertEqual "" "5x+5" (simplif "2x+1+3x+4")`

Tabela 4.5 Exemplos de construções de asserções com operadores

Sobrecarga de funções de asserções

Outra forma de construir asserções é utilizar a função com sobrecarga `assert` da classe de tipo `Assertable`:

```
class Assertable t
 where assert :: t -> Assertion

instance Assertable ()
 where assert = return

instance Assertable Bool
 where assert = assertBool ""
```

```
instance (ListAssertable t) => Assertable [t]
 where assert = listAssert

instance (Assertable t) => Assertable (IO t)
 where assert = (>>= assert)
```

A classe `ListAssertable` permite que a função `assert` possa ser aplicada a uma lista de caracteres `[Char]`, ou seja, uma *string*:

```
class ListAssertable t
 where listAssert :: [t] -> Assertion

instance ListAssertable Char
 where listAssert = assertString
```

Com as declarações indicadas é possível definir asserções que nunca falham, tais como:

```
assert ()
assert True
assert ""
```

Também é possível definir asserções que falham sempre, tais como:

```
assert False
assert "some failure message"
```

É possível definir outras instâncias para as classes de tipos `Assertable`, `ListAssertable` e `AssertionPredicable` que possam ser úteis no contexto de uma aplicação em particular.

Operadores de construções de testes

Podem ser utilizados os operadores seguintes para construir testes:

Operador	Teste criado
`pred ~? msg`	`TestCase (pred @? msg)`
`expected ~=? actual`	`TestCase (expected @=? actual)`
`actual ~?= expected`	`TestCase (actual @?= expected)`
`label ~: t`	`TestLabel label (test t)`

Tabela 4.6 Operadores de construção de testes

Os operadores `~?`, `~=?` e `~?=` construem uma asserção e um caso de teste para essa asserção. São utilizados da mesma forma que os operadores `@?`, `@=?` e `@?=` só que constroem imediatamente o teste.

O operador `~:` associa um nome (*label*) a um teste já existente.

No âmbito da aplicação de derivação e simplificação de expressões matemáticas, alguns testes podem ser escritos mais facilmente :

Operação	**Teste criado**
`(isNumber "2.5") ~? "isNumber 2.5"`	`TestCase ((isNumber "2.5") @? "isNumber 2.5")`
`"5x+5" ~=? (simplif "2x+1+3x+4")`	`TestCase ("5x+5" @=? (simplif "2x+1+3x+4"))`
`(simplif "2x+1+3x+4") ~?= "5x+5"`	`TestCase ((simplif "2x+1+3x+4") @?= "5x+5")`

Tabela 4.7 Exemplos de construções de testes com operadores

Sobrecarga de funções de testes

Outra forma de construir testes é utilizar a função com sobrecarga `test` da classe de tipo `Testable`:

```
class Testable t
 where test :: t -> Test

instance Testable Test
 where test = id

instance (Assertable t) => Testable (IO t)
 where test = TestCase . assert

instance (Testable t) => Testable [t]
 where test = TestList . map test
```

A função `test` constrói um teste tanto através de um valor de tipo `Assertion` (utilizando a função `TestCase`), uma lista de testes (utilizando `TestList`) ou um valor de tipo `Test` (não fazendo alterações).

Capítulo 5

Arquitectura do ambiente de testes

5.1 Componentes da arquitectura do ambiente de testes

Para o ambiente de testes desenhou-se uma arquitectura com quatro componentes, esquematizada na figura 5.1. O objectivo é disponibilizar ao programador uma plataforma de testes para programas em Haskell que se adapte a qualquer programa a ser testado. Os casos de teste são especificados através de ficheiros texto num dialecto XML criado para o efeito. Um componente designado *Testador* (que estende a *framework* HUnit com novas funcionalidades) executa os testes indicados no ficheiro e produz um ficheiro de saída em *plain text* com os resultados dos casos de teste. O componente *Interface* serve de intermediário entre o *Testador* e a aplicação a testar, evitando que dependam um do outro. Embora o *Testador* dependa de um *Interface*, este varia de aplicação para aplicação.

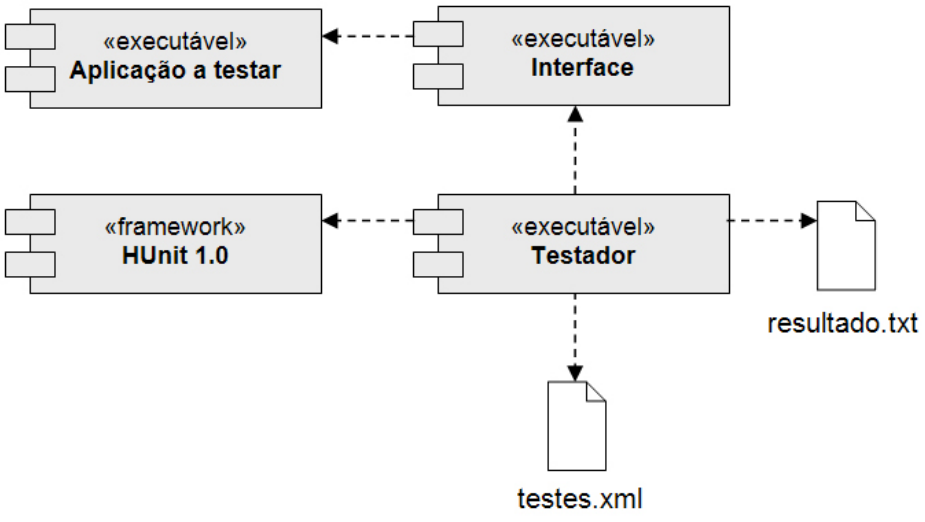

Figura 5.1 Arquitectura do ambiente de testes

Nas secções seguintes descreve-se cada um dos componentes da arquitectura de testes.

5.2 *Framework* HUnit

A HUnit é uma *framework* de testes unitários para a linguagem Haskell, desenvolvida em 2002 por Dean Herington. É um componente independente na arquitectura.

5.3 Aplicação a testar

Sendo o ambiente de testes a linguagem Haskell, a aplicação a testar terá que ser codificada nesta linguagem. Embora a aplicação possa ter interacção IO, os testes só poderão incidir em funções que não tenham estas interacções. Não existem outras restrições de especial à aplicação a testar, sendo esta uma componente independente na arquitectura.

5.4 Testador

O Testador destina-se a executar uma suite de testes extendendo a *framework* HUnit com novas funcionalidades, nomeadamente:

- Independência relativamente à aplicação a testar;
- Inclusão de um *parser* XML para HtestML, que permite escrita dos testes em ficheiro XML (em vez de directamente no código), permitindo facilmente a adopção de testes *data driven*;
- Resultados da execução dos testes escritos num ficheiro texto (em vez de terminal de saída);
- Tradução dos resultados para Português;
- Diferenciação de diferentes execuções do mesmo conjunto de testes.

5.5 Interface

O Interface associa o Testador à Aplicação, de forma a que um não dependa do outro. As suas principais funções são:

- Associar funções indicadas no ficheiro XML (sob a forma de *string*) com as funções a invocar da aplicação a testar;
- Fornecer instâncias da função `Show` (que apresenta resultados no terminal de saída) de forma a permitir que funções que manipulem valores diferentes de *string* possam ser incluídas nos casos de teste;

Para cada aplicação a testar tem que existir um componente Interface associado, pelo que este módulo depende sempre da aplicação. Por esse motivo, o Testador incluí um módulo Interface que pode ser sempre diferente. O Testador invoca a função `interpretar` do Interface para traduzir os testes indicados no ficheiro XML em funções presentes na aplicação a testar. É este mecanismo que permite que Testador não dependa directamente da aplicação e sim de uma Interface que conserva o seu nome mas varia conforme a aplicação.

5.6 Ficheiro de testes e dialecto HtestML

Para a definição dos casos de teste e a sua organização em hierarquias de teste, foi concebido um dialecto XML, designado de HtestML (*Haskell Test Markup Language*). A natureza hierárquica das árvores de teste torna o formato XML adequado para a sua representação. O Testador solicita um ficheiro XML e converte-o numa lista de testes em memória de acordo com a hierarquia indicada pela profundidade dos elementos no documento.

Com este formato é possível indicar:

- Nomes para testes
- Descrição opcional para casos de teste
- Função a executar em casos de teste
- Valor a avaliar na função indicada
- Valor a esperar na avaliação da função indicada

Para formalizar a especificação do formato, indica-se a descrição através de um DTD (*Document Type Definition*) do dialecto HtestML:

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT TESTE (NOME,(CASO_TESTE | TESTE)+)>
<!ELEMENT NOME(#PCDATA)>
<!ELEMENT CASO_TESTE (DESCRICAO+, FUNCAO, AVALIAR, ESPERAR)>
<!ELEMENT DESCRICAO (#PCDATA)>
<!ELEMENT FUNCAO(#PCDATA)>
<!ELEMENT AVALIAR (#PCDATA)>
<!ELEMENT ESPERAR(#PCDATA)>
```

Um formato mais completo para indicar a sintaxe do dialecto é o XSD (*XML Schema Definition*), que é mais rico em termos descrição de ocorrências e tipos de dados de elementos. Apresenta-se a seguir o *Schema* de descrição do dialecto HtestML:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
    <xs:element name="TESTE">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="NOME" type="xs:string"/>
                <xs:choice minOccurs="1" maxOccurs="unbounded">
                    <xs:element ref="CASO_TESTE"/>
                    <xs:element ref="TESTE"/>
                </xs:choice>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="CASO_TESTE">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="DESCRICAO" type="xs:string" minOccurs="0" maxOccurs="1"/>
                <xs:element name="FUNCAO" type="xs:string" minOccurs="1" maxOccurs="1"/>
                <xs:element name="AVALIAR" type="xs:string" minOccurs="1" maxOccurs="1"/>
                <xs:element name="ESPERAR" type="xs:string" minOccurs="1" maxOccurs="1"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Apresenta-se a seguir o documento que representa a árvore de testes referente à figura 4.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<TESTE xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"xsi:noNamespaceSchemaLocation="HtestML.xsd">
    <NOME>Polinómios</NOME>
    <TESTE>
        <NOME>Somas</NOME>
        <CASO_TESTE>
            <DESCRICAO>Números</DESCRICAO>
            <FUNCAO>simplif</FUNCAO>
            <AVALIAR>2+7</AVALIAR>
            <ESPERAR>9</ESPERAR>
        </CASO_TESTE>
        <CASO_TESTE>
            <DESCRICAO>Monómios</DESCRICAO>
            <FUNCAO>simplif</FUNCAO>
            <AVALIAR>2x+7x</AVALIAR>
            <ESPERAR>9x</ESPERAR>
        </CASO_TESTE>
        <CASO_TESTE>
            <DESCRICAO>Polinómio</DESCRICAO>
            <FUNCAO>simplif</FUNCAO>
            <AVALIAR>3x^2+9x^2+3x+5+x</AVALIAR>
            <ESPERAR>12x^2+4x+5</ESPERAR>
        </CASO_TESTE>
    </TESTE>
    <TESTE>
        <NOME>Produtos</NOME>
        <CASO_TESTE>
            <DESCRICAO>Números</DESCRICAO>
            <FUNCAO>simplif</FUNCAO>
            <AVALIAR>2*7</AVALIAR>
            <ESPERAR>15</ESPERAR>
        </CASO_TESTE>
        <CASO_TESTE>
            <DESCRICAO>Monómios</DESCRICAO>
            <FUNCAO>simplif</FUNCAO>
            <AVALIAR>2x*7x</AVALIAR>
            <ESPERAR>14x^2</ESPERAR>
        </CASO_TESTE>
        <CASO_TESTE>
            <DESCRICAO>Polinómio</DESCRICAO>
            <FUNCAO>simplif</FUNCAO>
            <AVALIAR>3x^2*9x^2*3x*5*x</AVALIAR>
            <ESPERAR>405x^6</ESPERAR>
        </CASO_TESTE>
    </TESTE>
    <CASO_TESTE>
        <FUNCAO>simplif</FUNCAO>
        <AVALIAR>2^3</AVALIAR>
        <ESPERAR>8</ESPERAR>
    </CASO_TESTE>
</TESTE>
```

5.7 Ficheiro de resultados

O Testador envia os resultados da execução dos testes indicados no documento HtestML para um ficheiro texto (*plain text*). O ficheiro de resultados indica o ficheiro de origem dos testes em HtestML, a data ocorreu a execução e o nº de vezes que esse ficheiro foi executado nessa data (identificando assim a execução em causa). O próprio nome que o Testador atribui ao ficheiro segue um formato que permite perceber estas atributos.

O formato adoptado para o nome dos ficheiros é o seguinte:

```
{ficheiro_origem}_{data}_{nº execucao}.txt
```

A parte restante do ficheiro consiste na apresentação do resultado da execução de cada caso de teste. Este ficheiro apresenta uma versão modificada do *output* produzido pela *framework* HUnit, com as mensagens traduzidas para Português. Tal como na *framework*, para os casos bem sucedidos são apresentadas somente os quatro contadores, no entanto, para os casos que apresentem falhas, é indicada a função, obtido do elemento `FUNCAO`, o valor esperado, obtido do elemento `ESPERADO`, e o valor obtido ao aplicar à função o valor indicado no elemento `AVALIAR`.

A indicação do caminho do caso de teste foi profundamente alterada. O caminho é especificado de acordo com o standard *XPath* do W3C, que identifica elementos num documento XML. Alguns editores de XML, como o *XML Spy*, aceitam expressões *XPath* e navegam directamente para o elemento XML identificado. Desta forma é possível facilmente identificar no documento que contem os testes, o caso de teste que falhou

Capítulo 6

Concepção dos casos de teste

6.1 Abordagem para a concepção dos casos de teste

A motivação para a concepção de casos de teste e a sua posterior execução tem dois objectivos. Por um lado é necessário verificar se a especificação é cumprida, através da detecção de *defeitos*. Os defeitos ocorrem quando uma funcionalidade não tem o funcionamento esperado. Por outro lado, é necessário medir a *qualidade dos resultados* obtidos pela aplicação. Um resultado, apesar de correcto, pode ter vários níveis de qualidade. Este aspecto é particularmente mensurável na aplicação em teste quando efectua simplificações de expressões.

Os casos de teste são concebidos com base na especificação da aplicação a testar. A especificação é expressa num conjunto de regras agrupadas por categorias. Cada regra tem um código, uma descrição e especifica o *input* e *output* esperados de uma forma abstracta.

O tipo de abordagem consiste nos seguintes pontos:

- Tratar cada regra (mais precisamente, a gama de valores de entrada especificada na condição) como uma classe de equivalência
- Uma vez que a especificação é recursiva, um mesmo caso de teste pode cobrir várias regras
- Encontrar um conjunto mínimo de casos de teste que cubram todas as regras
- Quando uma regra é testada mais do que uma vez, usar valores diferentes

Nas secções seguintes apresentam-se as regras de especificação e os casos de testes concebidos para as contemplar.

6.2 Regras de especificação

As regras de especificação da aplicação em teste – Aplicação de simplificação e derivação de expressões matemáticas – são apresentadas em três categorias, que correspondem às funcionalidades a testar:

- Conversão de formatos
- Simplificação de expressões
- Derivação de expressões

6.2.1 Conversão de formatos

A conversão de formatos engloba os seguintes duas categorias de conversão. As conversões entre expressões no formato interno e externo e as conversões entre expressões no formato interno e representação em árvore. Ambas as categorias tem conversores nos dois sentidos de conversão. Associada à conversão de expressões do formato interno para o externo está a operação de validação sintáctica.

Conversão de formato externo para formato interno

Cada regra deste conjunto indica uma transformação que é efectuada para a conversão de uma expressão originalmente do formato externo (conhecido pelo utilizador) para o formato interno (antes da conversão em árvore). As transformações associadas a cada regra vão sendo efectuadas pela ordem apresentada na tabela 6.1.

Cod.	Descrição	Formato externo	Formato interno
CVFI01	Remoção de espaços	$a \quad + b$	$a + b$
CVFI02	Conversão de maiúsculas em minúsculas	$Sen(a)$	$sen(a)$
CVFI03	Inserção de radical 2 onde implícito	$\sqrt{a}$	$\sqrt[2]{a}$
CVFI04	Inserção de sinal $\times$ onde implícito	$2a$	$2 * a$
CVFI05	Inserção de factor (-1) onde omitido	$2 - a$	$2 - 1 * a$
CVFI06	Inserção de sinal + antes de cada sinal $-$	$2 - 1 * a$	$2 + -1 * x$

Tabela 6.1 Transformação de expressões para o formato interno

As regras CVFI01 e CVFI02 são sempre efectuadas para qualquer expressão. As regras restantes, pelo contrário, só são efectuadas perante a presença de operadores específicos.

Validação sintáctica

No cenário da conversão de formatos é efectuada a validação sintáctica de expressões no formato interno. A validação sintáctica é descrita por um conjunto de regras às quais correspondem verificações efectuadas pela ordem apresentada na tabela 6.2:

Cod.	Descrição	Causa	Expressão
VSIN01	Expressão não indicada	Ausência de caracteres	
VSIN02	Símbolo inválido	Presença de caracteres não definidos na gramática	$2 : 4$
VSIN03	Parêntesis não fechados	Parêntesis abertos diferentes de parêntesis fechados	$2x((x + 1)$
VSIN04	Expressão inválida	Sequência gramatical não definida	$2 + *5$
VSIN05	Função desconhecida	Função não definida na gramática	$sec(x)$
VSIN06	Elemento desconhecido	Variável não definida na gramática	$2t + 1$

Tabela 6.2 Verificações efectuadas para validação sintáctica da expressão

Conversão de formato interno para formato externo

As regras desta secção dizem respeito às transformações inversas às apresentadas na tabela 6.1, um vez que se referem ao processo inverso. As regras correspondentes a essas transformações apresentam-se na tabela 6.3:

Cod.	Descrição	Formato interno	Formato externo
CVFE01	Remoção do radical 2 onde implícito	$\sqrt[2]{a}$	$\sqrt{a}$
CVFE02	Remoção do sinal $*$ onde implícito	$2 * a$	$2a$
CVFE03	Inserção de factor (-1) onde omitido	$2 - 1 * a$	$2 - a$
CVFE04	Inserção de sinal + antes de cada sinal −	$2 + -1 * x$	$2 - 1 * a$

Tabela 6.3 Transformação de expressões para o formato externo

Construção de árvore a partir de expressão

A segunda categoria de conversões são as que envolvem a transformação de expressões do formato interno para a sua representação em árvore e vice-versa. A aplicação efectua a análise léxica dos constituintes de uma expressão formando uma representação em árvore (*parse tree*) cujos nós são elementos. Cada elemento é um tuplo formado por quatro atributos, de acordo com a seguinte organização:

```
Elemento (Tipo, Coeficiente, Expoente, [sub-elementos])
```

Cada elemento tem um tipo (valor alfanumérico), um coeficiente e expoente (valores numéricos) e uma lista de sub-elementos (que pode ser vazia). A tabela 6.4 mostra os atributos dos diferentes tipos de elementos considerados pela aplicação:

Cod.	Descrição	Tipo	Coeficiente	Expoente	Sub-elementos
REPA01	Constante	k	*valor*	0	—
REPA02	Variável	x	1	1	—
REPA03	Adição	$+$	1	1	Parcelas
REPA04	Multiplicação	$*$	1	1	Factores
REPA05	Potenciação	$\wedge$	1	1	Base e Expoente
REPA06	Função	*nome*	1	1	Argumento

Tabela 6.4 Construção da representação em árvore de uma expressão

Para construir a árvore, é necessário separar os operandos dos operadores na expressão. Para esse efeito, é detectada a operação de menor precedência e as posições em ocorre na expressão. A presença de parêntesis implica que a expressão que encapsulam não é considerada para a detecção da operação de menor prioridade da expressão exterior.

6.2.2 Simplificação de Expressões

A funcionalidade de simplificação de expressões constitui um dos objectivos da aplicação, sendo efectuada pelo bloco funcional designado por *Simplificador*.

Operadores algébricos básicas

O primeiro grupos de regras de especificação do processo de simplificação são a descrição das operações algébricas básicas suportadas, apresentadas na tabela 6.5:

Cod.	Operação	Descrição	Expressão
OPER01	Adição	c é o resultado da soma de a com b	$c = a + b$
OPER01	Multiplicação	c é o resultado do produto de a com b	$c = a \times b$
OPER01	Potenciação	c é o resultado de elevar a a b	$c = a^b$

Tabela 6.5 Operações algébricas básicas tratadas pela aplicação

Propriedades dos operadores algébricos

Para estas operações devem verificar as propriedades dos operadores algébricos, apresentadas na tabela 6.6 e que também definem outros operadores:

Cod.	Descrição	Propriedade
PALG01	Elemento neutro da adição	$a + 0 = a$
PALG02	Elemento inverso da adição	$a + (-a) = 0$
PALG03	Comutatividade da adição	$a + b = b + a$
PALG04	Associatividade da adição	$(a + b) + c = a + (b + c)$
PALG05	Definição de subtracção	$a - b = a + (-b)$
PALG06	Elemento neutro da multiplicação	$a * 1 = a$
PALG07	Elemento absorvente da multiplicação	$a * 0 = 0$
PALG08	Elemento inverso da multiplicação	$a * \frac{1}{a} = 1$
PALG09	Comutatividade da multiplicação	$a * b = b * a$
PALG10	Associatividade da multiplicação	$(a * b) * c = a * (b * c)$
PALG11	Definição de divisão	$\frac{a}{b} = a * (\frac{1}{b})$
PALG12	Soma de expoentes	$a^b \times a^c = a^{(b+c)}$
PALG13	Produto de bases	$a^c \times b^c = (a + b)^c$
PALG14	Produto de expoentes	$(a^b)^c = a^{b*c}$
PALG15	Expoente inverso	$a^{-b} = \frac{1}{a^b}$
PALG16	Diferença de expoentes	$a^{b-c} = \frac{a^b}{a^c}$
PALG17	Definição de radiciação	$\sqrt[c]{a^b} = a^{(\frac{b}{c})}$

Tabela 6.6 Propriedades dos operadores algébricos

Funções matemáticas

Finalmente, o processo de simplificação deve tratar algumas funções contempladas pela aplicação. Quando o argumento for numérico, deve ser substituído pelo valor correspondente à aplicação da função, caso contrário, deve ser simplificado o mais possível. A tabela 6.7 apresenta as funções contempladas pela aplicação, agrupadas por categorias. A descrição é comum a todas as regras, sendo b o resultado da aplicação da função ao seu argumento a, em que a é uma qualquer expressão. É de particular interesse a identificação do domínio de cada função, porque essa informação é importante para uma das regras de validação semântica.

Cod.	**Função**	**Expressão**	**Domínio** (D)
FUNC01	Seno	$b = sen(a)$	$a \in \mathbb{R}$
FUNC02	Co-seno	$b = cos(a)$	$a \in \mathbb{R}$
FUNC03	Tangente	$b = tg(a)$	$a \in \mathbb{R} : a \neq \frac{\pi}{2} + k\pi, k \in \mathbb{Z}$
FUNC04	Arco seno	$b = arcsen(a)$	$a \in [-1, 1]$
FUNC05	Arco co-seno	$b = arccos(a)$	$a \in [0, \pi]$
FUNC06	Arco tangente	$b = arctg(a)$	$a \in \mathbb{R}$
FUNC07	Logarítmica	$b = log(a)$	$a \in \mathbb{R}^+$
FUNC08	Exponencial	$b = exp(a)$	$a \in \mathbb{R}$

Tabela 6.7 Funções matemáticas tratadas pela aplicação

Validação semântica

Em simultâneo com o processo de simplificação de uma expressão é efectuada a sua validação semântica, onde se verificam as regras apresentadas na tabela 6.8. Todas as regras se referem a violações de domínio das expressões, no entanto, separam-se em três regras diferentes para identificar os casos de maior destaque.

Cod.	**Descrição**	**Causa**	**Expressão**
VSEM01	Argumento inválido	Violação do domínio de uma função da tabela 6.7	$a \notin D$
VSEM02	Divisão por zero	Denominador de fracção simplifica para zero	$\frac{a}{b}, b = 0$
VSEM03	Radicando negativo	Radicando negativo em raiz de índice par	$\sqrt[b]{a}, a < 0 \wedge b$ par

Tabela 6.8 Verificações efectuadas para validação semântica

6.2.3 Derivação de Expressões

A funcionalidade de derivação de expressões constitui o outro objectivo fundamental da aplicação em teste, sendo efectuada pelo bloco funcional designado por *Derivador*. As regras de especificação correspondem às regras de derivação suportadas pela aplicação e contemplam constantes, variáveis, as operações da tabela 6.5 e as funções da tabela 6.7.

A tabela 6.9 apresenta as regras de derivação suportadas pela aplicação. A notação utilizada é a seguinte:

- y representa a expressão a derivar
- y' representa a expressão referente ao cálculo da derivada de y
- a e b representam sub-expressões em y
- a' e b' representam as derivadas de a e b respectivamente

Cod.	Descrição	Expressão	Derivada
RDEV01	Constantes	$y = k$	$y' = 0$
RDEV02	Variáveis	$y = x$	$y' = 1$
RDEV03	Adição	$y = a + b$	$y' = a' + b'$
RDEV04	Multiplicação	$y = ab$	$y' = a'b + ab'$
RDEV05	Potenciação	$y = a^b$	$y' = ba^{b-1}a' + b'a^b log(a)$
RDEV06	Função seno	$y = sen(a)$	$y' = a'cos(a)$
RDEV07	Função co-seno	$y = cos(a)$	$y' = -a'sen(a)$
RDEV08	Função tangente	$y = tg(a)$	$y' = 1 + tg(a)^2$
RDEV09	Função arco seno	$y = arcsen(a)$	$y' = \frac{a'}{\sqrt{1-a}}$
RDEV10	Função arco co-seno	$y = arccos(a)$	$y' = -\frac{a'}{\sqrt{1-a}}$
RDEV11	Função arco tangente	$y = arctg(a)$	$y' = \frac{a'}{1+a}$
RDEV12	Função exponencial	$y = e^a$	$y' = a'e^a$
RDEV13	Função logaritmo	$y = log(a)$	$y' = \frac{a'}{a}$

Tabela 6.9 Regras de derivação de expressões

6.3 Casos de Teste

A partir das regras de especificação da aplicação em teste são definidos um conjunto de casos de teste. Os casos de teste devem cobrir todas as regras tendo como objectivo, por um lado, verificar a conformidade da aplicação com a especificação e por outro, medir a qualidade dos resultados obtidos.

Os casos de teste são definidos por categorias, seguindo uma estrutura semelhante aos grupos de regras definidos na secção anterior. Por esse motivo, os testes estão agrupados em três grandes grupos: testes de conversão de formatos, testes de simplificação e testes de derivação. Os testes de conversão de formatos incidem na verificação das regras apresentadas nas tabelas 6.1, 6.2, 6.3 e 6.4. Os testes de simplificação de expressões incidem na verificação das regras apresentadas nas tabelas 6.5, 6.6, 6.7 e 6.8 e também na medição da qualidade dos resultados obtidos. Os testes de derivação de expressões incidem na verificação das regras apresentadas na tabela 6.9 e tal como no grupo anterior, na medição da qualidade dos resultados obtidos.

6.3.1 Conversão de formatos

Testes de conversão entre o formato externo e o formato interno

Os teste de conversão de formato externo para formato interno envolvem a execução da função da interface de testes `ivalidar_sint`, que não tem interacção IO, efectuando a conversão e a validação sintáctica. Como *input* indica-se a expressão no formato externo e como *output* obtém-se a expressão correspondente no formato interno.

1. **Testes de integridade de valores numéricos**
   A integridade de valores numéricos em expressões é um aspecto não contemplado nas regras de especificação. Não há informação sobre a gama de valores admitida para as constantes, nem para o número máximo de casas decimais. Este grupo de testes tem como objectivo verificar se a conversão para o formato interno não afecta constantes, e em simultâneo, se a aplicação consegue tratar gamas de valores elevadas, bem como um grande número de casas decimais. Não estando definida na gramática das expressões a notação cientifica para exprimir valores de gamas elevadas, nos casos de teste é necessário a indicar estes valores em notação normal. Note-se que nesta conversão não ocorre qualquer simplificação de valores, pelo que é de esperar que os valores se mantenham com o mesmo aspecto, ainda que possam conter dígitos redundantes.

N°	***Input***	***Output***	**Descrição**
1	0	0	Zero sem sinal com um digito
2	0000	0000	Zero sem sinal com dígitos redundantes
3	1	1	Valor inteiro positivo com um digito
4	0001	0001	Valor inteiro positivo com dígitos redundantes
5	123456	123456	Valor inteiro positivo com vários dígitos
6	10000000000	10000000000	Valor inteiro positivo elevado
7	-0	-0	Zero com sinal negativo com um digito
8	-0000	-0000	Zero com sinal negativo com dígitos redundantes
9	-1	-1	Valor inteiro negativo com um digito
10	-0001	-0001	Valor inteiro negativo com dígitos redundantes
11	-123456	-123456	Valor inteiro negativo com vários dígitos
12	-10000000000	-10000000000	Valor inteiro negativo elevado
13	0.0	0.0	Zero sem sinal com um digito com uma casa decimal
14	0000.0000	0000.0000	Zero sem sinal com dígitos redundantes e casas decimais
15	1.5	1.5	Valor inteiro positivo com um digito e uma casa decimal
16	456.7890123	456.7890123	Valor inteiro positivo com casas decimais
17	-0.0	-0.0	Zero com sinal negativo com uma casa decimal
18	-1000.0000	-1000.0000	Valor inteiro negativo com casas decimais redundantes
19	-1.5	-1.5	Valor inteiro negativo com um digito com uma casa decimal
20	-456.7890123	-456.7890123	Valor inteiro negativo com casas decimais

Tabela 6.10 Testes de integridade de valores numéricos inteiros

2. **Testes de transformação de expressões para o formato interno**
   Este grupo de testes verifica uma a uma as regras da tabela 6.1 isoladas e depois combinadas numa só expressão.

N°	***Input***	***Output***	**Cobre regra CVFI**
1	$X \quad +1 \quad + SEN(Cos(x))$	$x + 1 + sen(cos(x))$	01 e 02
2	$\|(x+2)+3\|x$	$2\|(x+2)+3\|x$	03
3	$2x + xsen(x)$	$2 * x + x * sen(x)$	04
4	$x(x+1)(x+2)$	$x * (x+1) * (x+2)$	04
5	$2 - x + 5 - 3$	$2 + -1 * x + 5 + -3$	05 e 06
6	$-(x+1) \wedge 2$	$-1 * (x+1) \wedge 2$	05 e 06
7	$\mid (x - (X \wedge 5cos(x)))$	$2\|(x + -1 * (x \wedge 5 * cos(x)))$	01 a 06

Tabela 6.11 Testes de conversão de formato externo para o formato interno

3. **Testes de validação sintáctica**
   Este grupo de testes verifica a correcta detecção dos erros de validação sintáctica apresentados na tabela 6.2 isolados. Não faz muito sentido um caso de teste que combine os vários erros porque a validação, perante um erro, já não detecta os seguintes, acusando somente o primeiro.

N°	***Input***	***Output***	**Cobre regra VSIN**
1		ERRO: Funcao nao indicada	01
2	2?3	ERRO: Simbolo invalido	02
3	$cos(2 \setminus 4)$	ERRO: Simbolo invalido	02
4	$y = 2x + 3$	ERRO: Simbolo invalido	02
5	$((x+1)(x+2)+3$	ERRO: Parenteses nao fechados correctamente	03
6	$(x+1))(x+2)+3$	ERRO: Parenteses nao fechados correctamente	03
7	$sen(x+3$	ERRO: Parenteses nao fechados correctamente	03
8	$+1$	ERRO: Expressao invalida	04
9	$*1$	ERRO: Expressao invalida	04
10	$2 + *5$	ERRO: Expressao invalida	04
11	$2 + sec(x)$	ERRO: Funcao 'sec' desconhecida	05
12	$sen(cotg(x))$	ERRO: Funcao 'cotg' desconhecida	05
13	$2 + sec + x$	ERRO: Elemento 'sec' desconhecido	06
14	$sen(t)$	ERRO: Elemento 't' desconhecido	06
15	$2x + 3$ *equation*	ERRO: Elemento 'equation' desconhecido	06

Tabela 6.12 Testes de conversão de validação sintáctica

4. **Testes de transformação de expressões para o formato externo**
   Este grupo de testes verifica as regras da tabela 6.1 isoladas, mas na direcção contrária. Excluindo as duas primeiras que não têm contrapartida para a transformação inversa, verificam-se

a recuperação do formato externo a partir de uma expressão no formato interno.

N°	*Input*	*Output*	Cobre regra CVFI
1	$2\|(x+2)+3\|x$	$\|(x+2)+3\|x$	03
2	$2*x+x*sen(x)$	$2x+xsen(x)$	04
3	$x*(x+1)*(x+2)$	$x(x+1)(x+2)$	04
4	$2+-1*x+5+-3$	$2-x+5-3$	05 e 06
5	$-1*(x+1)\wedge 2$	$-(x+1)\wedge 2$	05 e 06
6	$2\|(x+-1*(x\wedge 5*cos(x)))$	$\|\ (x-(X\wedge 5cos(x)))$	01 a 06

Tabela 6.13 Testes de conversão de formato interno para o formato externo

Testes de conversão entre o formato interno e a representação em árvore

Os testes de conversão de formato interno para representação em árvore envolvem a execução da função da interface `iespandir`. A função `espandir` efectua na aplicação a conversão de uma expressão em *string* para a sua representação em árvore. Esta função levantou o problema da apresentação dos resultados, já que o tipo de dados de saída é `Expressao`. Para tornar o resultado de saída como uma string e assim possibilitar a sua utilização na arquitectura de testes definida, foi necessário incluir na interface de testes um instância da função `show` para o tipo de dados `Expressao`. A função `iespandir` da interface por sua vez invoca a função `show` sobre a função `espandir` o que possibilita a apresentação do resultado como uma *string*.

Como *input* indica-se a expressão no formato interno e como *output* obtém-se a lista correspondente à representação em árvore.

1. **Testes básicos de representação em árvore**
   Os primeiros testes a efectuar cobrem de uma forma básica as regras de criação de elementos que constituem nós da representação em árvore da expressão.

N°	*Input*	*Output*	Cobre regra REPA
1	-3.5	$(k,-3.5,0.0,[])$	01
2	x	$(x,1.0,1.0,[])$	02
3	$2+x$	$(+,1.0,1.0,[(k,2.0,0.0,[])(x,1.0,1.0,[])])$	01, 02 e 03
4	$2*x$	$(*,1.0,1.0,[(k,2.0,0.0,[])(x,1.0,1.0,[])])$	01, 02 e 04
5	$2\wedge x$	$(\wedge,1.0,1.0,[(k,2.0,0.0,[])(x,1.0,1.0,[])])$	01, 02 e 05
6	$cos(x+2)$	$(cos,1.0,1.0,[(+,1.0,1.0,[(x,1.0,1.0,[])(k,2.0,0.0,[])])])$	01, 02, 03 e 06

Tabela 6.14 Testes básicos de conversão de expressões no formato interno a representação em árvore

2. **Testes genéricos de representação em árvore**
   O grupo de testes na tabela 6.15 procuram cobrir várias regras numa mesma expressão de forma a aferir a complexidade da árvore gerada.

N°	*Input/Output*
1	$2+3+-4+5$
	$(+,1.0,1.0,[(k,2.0,0.0,[])(k,3.0,0.0,[])(k,-4.0,0.0,[])(k,5.0,0.0,[])])$
2	$2*3+-4*x+5$
	$(+,1.0,1.0,[(*,1.0,1.0,[(k,2.0,0.0,[])(k,3.0,0.0,[])])$
	$(*,1.0,1.0,[(k,-4.0,0.0,[])(x,1.0,1.0,[])])(k,5.0,0.0,[])])$
3	$4*x \wedge 2+3*x+1$
	$(+,1.0,1.0,[(*,1.0,1.0,[(k,4.0,0.0,[])(\wedge,1.0,1.0,[(x,1.0,1.0,[])(k,2.0,0.0,[])])])$
	$(*,1.0,1.0,[(k,3.0,0.0,[])(x,1.0,1.0,[])])(k,1.0,0.0,[])])$
4	$(x+2.5) \wedge 3.2$
	$(\wedge,1.0,1.0,[(+,1.0,1.0,[(x,1.0,1.0,[])(k,2.5,0.0,[])])(k,3.2,0.0,[])])$
5	$2*x*(3+x+5)$
	$(*,1.0,1.0,[(k,2.0,0.0,[])(x,1.0,1.0,[])(+,1.0,1.0,[(k,3.0,0.0,[])(x,1.0,1.0,[])(k,5.0,0.0,[])])])$
6	$(x+1)*(x+2)$
	$(*,1.0,1.0,[(+,1.0,1.0,[(x,1.0,1.0,[])(k,1.0,0.0,[])])(+,1.0,1.0,[(x,1.0,1.0,[])(k,2.0,0.0,[])])])$
7	$((x+1))$
	$(+,1.0,1.0,[(x,1.0,1.0,[])(k,1.0,0.0,[])])$
8	$((((x+1))))$
	$(+,1.0,1.0,[(x,1.0,1.0,[])(k,1.0,0.0,[])])$
9	$((((x+1))+2))$
	$(+,1.0,1.0,[(+,1.0,1.0,[(x,1.0,1.0,[])(k,1.0,0.0,[])])(k,2.0,0.0,[])])$
10	$tg(log(2*x)+sen(x))$
	$(tg,1.0,1.0,[(+,1.0,1.0,[(log,1.0,1.0,[(*,1.0,1.0,[(k,2.0,0.0,[])(x,1.0,1.0,[])])])$
	$(sen,1.0,1.0,[(x,1.0,1.0,[])])])])$

Tabela 6.15 Testes genéricos de conversão de expressões no formato interno a representação em árvore

6.3.2 Simplificação de Expressões

Os testes de simplificação de expressões serão efectuados sob um ponto de vista de testes de sistema. Os *inputs* serão expressões no formato externo, que serão convertidas para o formato interno e posteriormente para sua representação em árvore. Nessa representação, é aplicada a função de simplificação que reduz a árvore o mais possível. A árvore é convertida na expressão no formato interno e de seguida no formato externo, que serão os *outputs*.

Os testes podiam incidir somente na componente de simplificação, no entanto, tal obrigaria a trabalhar com representações em árvore que são de escrita extensa e pouco cómoda, não trazendo qualquer mais-valia para os objectivos destes testes. Estes são por um lado, avaliar a correcção da expressão simplificada e por outro medir a qualidade da simplificação, ou seja, se de facto a expressão resultante é a mais simples possível ou se seria possível efectuar mais simplificações.

Os testes de simplificação incluem verificação das regras e propriedades dos operadores algébricos básicas, das funções definidas, regras de validação semântica, requisitos básicos de simplificação e por fim teste de qualidade.

Testes de simplificação de operações algébricas básicas

O grupo de testes da tabela 6.16 destina-se a verificar expressões que envolvam cálculo numérico de constantes em diferentes contextos (como constantes e coeficientes), nas três operações algébricas básicas (adição, multiplicação e potenciação).

N°	***Input***	***Output***	**Cobre regra OPER**
1	$1.1 + 2.3 + 4$	7.4	01
2	$1.1 * 2.3 * 4$	10.12	02
3	$1.1 \wedge 2.3 \wedge 4$	14.399156128283	03
4	$3 * 5 + 2 \wedge 3 + 1$	24	01, 02 e 03
5	$3x \wedge 2 + 2x + 4 + 5x \wedge 2 + 3x + 2$	$8x \wedge 2 + 5x + 6$	01, 02 e 03
6	$sen(x) + 4 + 3x + 2sen(x) + 2 + x$	$3sen(x) + 6 + 4x$	01, 02 e 03

Tabela 6.16 Testes de operações algébricas básicas

Testes de verificação de propriedades do operadores algébricos

O grupo de testes das tabelas 6.17 e 6.18 permitem verificar as propriedades dos operadores algébricos apresentadas nas regras de especificação da tabela 6.6.

N°	***Input***	***Output***	**Cobre regra PALG**
1	$5 + 0 + 3$	8	01
2	$5 + (-5)$	0	02
3	$2 + x + 3$	$x + 5$	03
4	$x + 2 + 3$	$x + 5$	03
5	$(2 + x) + 3$	$x + 5$	04
6	$2 + (x + 3)$	$x + 5$	04
7	$2 - 3$	-1	05
8	$2 + (-3)$	-1	05
9	$5 * 1 * 3$	15	06
10	$5 * 0 * 3$	0	07
11	$3x$	$3x$	09
12	$x * 3$	$3x$	09
13	$(2x) * 3$	$6x$	10
14	$2(3x)$	$6x$	10
15	$2/5$	0.4	11

Tabela 6.17 Testes das propriedades dos operadores algébricos

N°	*Input*	*Output*	Cobre regra PALG
16	$2 * 0.2$	0.4	11
17	$5 * (1/5)$	1	08
18	$3 \wedge 2 * 3 \wedge 3$	243	12
19	$3 \wedge (2 + 3)$	243	12
20	$2 \wedge 2 \wedge 3$	256	14
21	$2 \wedge (2 * 3)$	256	14
22	$(2 \wedge 2) \wedge 3$	64	14
23	$2 \wedge (-3)$	0.125	15
24	$1/(2 \wedge 3)$	0.125	15
25	$3 \wedge (5 - 2)$	27	16
26	$3 \wedge 5/3 \wedge 2$	27	16
27	$\|(3 \wedge 4)$	9	17
28	$3 \wedge (4/2)$	9	17

Tabela 6.18 Testes das propriedades dos operadores algébricos

Testes de simplificação de avaliação de funções

O grupo de testes das tabelas 6.19 e 6.20 permitem verificar as situações fronteira da avaliação das funções trigonométricas suportadas pela aplicação. Algumas dessas situações são valores conhecidos que permitem facilmente determinar se as funções são correctamente avaliadas.

N°	*Input*	*Output*	Cobre regra FUNC
1	$sen(0)$	0	01
2	$sen(arcsen(1) * 2/3)$	0.866025403784439	01, 04
3	$sen(arcsen(1))$	1	01, 04
4	$cos(0)$	1	02
5	$cos(arcsen(1)/6)$	0.965925826289068	02, 04
6	$cos(arcsen(1))$	0	02, 04
7	$tg(0)$	0	03
8	$tg(arcsen(1) * 2/3)$	1.73205080756888	03, 04
9	$arcsen(0)$	0	04
10	$arcsen(1)$	1.5707963267949	04
11	$arccos(0)$	1.5707963267949	05

Tabela 6.19 Testes de simplificação de avaliação de funções

Testes de validação semântica

O grupo de testes das tabelas 6.21 destinam-se a verificar as regras de validação semântica através de expressões que contenham violações do domínio das funções envolvidas. As funções são as indicadas

N°	*Input*	*Output*	Cobre regra FUNC
12	$arccos(1)$	0	05
13	$arctg(0)$	0	06
14	$arctg(1)$	0.785398163397448	06
15	$e(0)$	1	08
16	$e(1)$	2.71828182845905	08
17	$log(1)$	0	07
18	$log(e(1))$	1	07, 08
19	$e(log(1))$	1	07, 08

Tabela 6.20 Testes de simplificação de avaliação de funções

na 6.7 que incluí uma coluna para a gama de valores admitidos no domínio. Nos testes seguintes, a violações de domínio é explicita em alguns casos e implícita noutros (advém de simplificação de valores).

N°	*Input*	*Output*	Cobre regra VSEM
1	$tg(arccos(-1)/2)$	ERRO: Argumento invalido em 'tg'	01
2	$arcsen(2)$	ERRO: Argumento invalido em 'arcsen'	01
3	$arccos(2 * arccos(-1))$	ERRO: Argumento invalido em 'arccos'	01
4	$log(0)$	ERRO: Argumento invalido em 'log'	01
5	$(5 * x)/(-3 + 8 - 5)$	ERRO: Divisao por zero	02
6	$3/(x - x)$	ERRO: Divisao por zero	02
7	$\|(-1)$	ERRO: Radicando negativo em raiz de radical par	03
8	$4\|(-2)$	ERRO: Radicando negativo em raiz de radical par	03

Tabela 6.21 Testes de validação semântica

Testes de qualidade de simplificação

O último conjunto de testes no âmbito da simplificação visa averiguar a qualidade da simplificação. Para estes testes, assume-se que a expressão em causa será simplificada sem erros. Nestes testes não são indicadas as regras de especificação cobertas, uma vez que não é esse o foco de análise. O foco da análise vira-se para a capacidade da aplicação em obter expressões o mais simples possível. Assim os resultados destes, indicam não se a expressão está correcta, mas se está efectivamente na sua forma mais simples.

1. Simplificação de Somas
   As expressões que envolvem somas foram agrupadas em dois conjuntos: o das somas simplificáveis (tabela 6.22) e o das somas não simplificáveis (tabela 6.23).

2. Simplificação de Diferenças
   Neste conjunto de testes (tabela 6.24) pretende-se avaliar se uma expressão simplifica correctamente para o valor zero através de diferenças de valores simétricos.

N°	*Input*	*Output*
1	$x \wedge 2 + 3x + 4 + 5x \wedge 2 + 4x + 6$	$6x \wedge 2 + 7x + 10$
2	$4 + 3x + x \wedge 2 + 5x \wedge 2 + 4x + 6$	$6x \wedge 2 + 7x + 10$
3	$(x \wedge 2 + 3x + 4) + (5x \wedge 2 + 4x + 6)$	$6x \wedge 2 + 7x + 10$
4	$(x \wedge 2 + 3x + 4) + 2(5x \wedge 2 + 4x + 6)$	$x \wedge 2 + 3x + 4 + 2(5x \wedge 2 + 4x + 6)$
5	$sen(x) + sen(x + 1) + sen(x) + sen(x + 1)$	$2sen(x) + 2sen(x + 1)$

Tabela 6.22 Testes de somas simplificáveis

N°	*Input*	*Output*
1	$x \wedge 2 + 3x + 4$	$x \wedge 2 + 3x + 4$
2	$4 + 3x + x \wedge 2$	$x \wedge 2 + 3x + 4$
3	$(x \wedge 2 + 3x) + 4$	$x \wedge 2 + 3x + 4$
4	$x \wedge 2 + (3x + 4)$	$x \wedge 2 + 3x + 4$
5	$(x \wedge 2 + 3x + 4)$	$x \wedge 2 + 3x + 4$
6	$sen(x) + cos(x)$	$sen(x) + cos(x)$
7	$sen(x) + sen(x) \wedge 2$	$sen(x) + sen(x) \wedge 2$
8	$sen(x \wedge 2) + sen(x)$	$sen(x \wedge 2) + sen(x)$
9	$sen(x) + (x + 1)$	$sen(x) + x + 1$

Tabela 6.23 Testes de somas não simplificáveis

3. Simplificação de Produtos
   As expressões que envolvem produtos foram agrupadas em dois conjuntos: o dos produtos simplificáveis (tabela 6.25) e o dos produtos não simplificáveis (tabela 6.26).

4. Simplificação de Divisões
   Neste conjunto de testes (tabela 6.27) pretende-se avaliar se uma expressão simplifica correctamente para o valor unitário ou se elimina factores neutros.

5. Simplificação de Potências
   As expressões que envolvem potências foram agrupadas em dois conjuntos: o das potências simplificáveis (tabela 6.28) e o das potências não simplificáveis (tabela 6.29).

6.3.3 Derivação de Expressões

Os testes de derivação de expressões são, à semelhança dos testes de simplificação, considerados testes de sistema. A sua execução é efectuada através da função `iderivar` que sequência todo o processo de derivação com as simplificações envolvidas. Num primeiro grupo de testes são verificadas as regras de derivação, sendo então efectuados testes com a composição de várias regras em expressões que dão origem a derivadas progressivamente mais complexas.

N°	*Input*	*Output*
1	$(x+1)-x-1$	0
2	$(x+1)-(x+1)$	0
3	$(x+1)\wedge 2-(x+1)\wedge 2$	0
4	$x\wedge 2+3x+4-5x\wedge 2-4x-6+4x\wedge 2+x+2$	0
5	$sen(x)+2sen(x)-3sen(x)$	0
6	$sen(x\wedge 2)-sen(x\wedge 2)$	0
7	$sen(x)\wedge 2-sen(x)\wedge 2$	0

Tabela 6.24 Testes de diferenças

N°	*Input*	*Output*
1	$2xxx$	$2x\wedge 3$
2	$x(x+1)(x+1)(x+2)$	$(x+1)\wedge 2x(x+2)$
3	$(x+1)x(x+1)(x+2)$	$(x+1)\wedge 2x(x+2)$
4	$(x+1)\wedge 2(x+2)(x+1)\wedge 3$	$(x+2)((x+1)\wedge 5)$
5	$sen(x)cos(x)sen(x)$	$sen(x)\wedge 2cos(x)$
6	$sen(x)x*cos(x)sen(x)\wedge 2$	$sen(x)\wedge 3*x*cos(x)$

Tabela 6.25 Testes de produtos simplificáveis

Regras de derivação

O grupo de testes da tabela 6.30 pretende verificar as regras de derivação apresentadas na tabela 6.9.

Composição de regras de derivação em expressões

O grupo de testes da tabela 6.31 pretende verificar o trabalho do processo de derivação e simplificação na produção de uma expressão. Os resultados a obter destes testes merecem uma análise crítica pois não são passíveis de poderem ser facilmente classificados como correctos ou errados. As expressões são simplificadas, de seguida derivadas e por fim a própria derivada é também simplificada. Basta que um termo numa expressão seja simplificado de forma diferente da esperada para que o teste respondente falhe, contudo tal não implica que a expressão não tenha sido correctamente simplificada.

N°	*Input*	*Output*
1	$x(x+1)$	$x(x+1)$
2	$x(x+1)(x+3)(x+2)$	$x(x+1)(x+3)(x+2)$
3	$x(x+1)(x+2)(x+3)$	$x(x+1)(x+2)(x+3)$
4	$sen(x+1)cos(x)sen(x)$	$sen(x+1)cos(x)sen(x)$
5	$sen(x)sen(x \wedge 2)$	$sen(x)sen(x \wedge 2)$

Tabela 6.26 Testes de produtos não simplificáveis

N°	*Input*	*Output*
1	$x(1/x)$	1
2	$x(x+1)/x$	$x+1$
3	$(x+1)x/x$	$x+1$
4	$(x+1)(x+2)/(x+1)$	$x+2$
5	$(x+2)(x+1)/(x+1)$	$x+2$
6	$((x+1)(x+2))/(x+1)$	$x+2$

Tabela 6.27 Testes de divisões

N°	*Input*	*Output*
1	$x \wedge 2 * x \wedge 3$	$x \wedge 5$
2	$x \wedge (x+1) + x \wedge (x+1)$	$2x \wedge (x+1)$
3	$x \wedge (x \wedge 2)$	$x \wedge x \wedge 2$
4	$x \wedge (x+1-1)$	$x \wedge x$
5	$x \wedge 8 + x \wedge 8$	$2x \wedge 8$
6	$x \wedge 2 \wedge 3 + x \wedge 8$	$2x \wedge 8$

Tabela 6.28 Testes de potências simplificáveis

N°	*Input*	*Output*
1	$x \wedge (x+1)$	$x \wedge (x+1)$
2	$(x+1) \wedge x$	$(x+1) \wedge x$
3	$x \wedge sen(x)$	$x \wedge sen(x)$
4	$sen(x) \wedge x$	$sen(x) \wedge x$

Tabela 6.29 Testes de potências não simplificáveis

N°	***Input***	***Output***	**Cobre regra RDEV**
1	3.5	0	01
2	x	1	02
3	$3x$	3	04
4	$x(4x+5)$	$8x+5$	04
5	$x \wedge (x+1)$	$x \wedge x(x+1) + log(x)x \wedge (x+1)$	05
6	$sen(x)$	$cos(x)$	06
7	$cos(x)$	$-sen(x)$	07
8	$tg(x)$	$1 + tg(x) \wedge 2$	08
9	$arcsen(x)$	$1/\|(1 + x \wedge 2)$	09
10	$arccos(x)$	$-1/(-(1 + x \wedge 2) \wedge 0.5)$	10
11	$arctg(x)$	$1/(1 + x \wedge 2)$	11
12	$e(x)$	$e(x)$	12
13	$log(x)$	$1/x$	13

Tabela 6.30 Testes de derivação de expressões

N°	*Input*	*Output*
1	$3x \wedge 2 - 5x + 1$	$6x - 5$
2	$(x + 3) \wedge 5$	$5(x + 3) \wedge 4$
3	$(x - 1)(x - 3)$	$2x - 4$
4	$(2x - 3)(x - 2)(x + 3)$	$6x \wedge 2 - 2x - 15$
5	$1/(x + 3)$	$-1/(x + 3) \wedge 2$
6	$(3x + 1)/(x \wedge 2 - 8x)$	$(-3x \wedge 2 - 2x + 8)/(x \wedge 2 - 8x) \wedge 2$
7	$((x - 1)/(x + 2)) \wedge 2$	$6(x - 1)/((x + 2) \wedge 3)$
8	$((x \wedge 2 - 1)/2x) \wedge 3$	$3(x \wedge 2 - 1) \wedge 2(x \wedge 2 + 1)/(8x \wedge 4)$
9	$\|(x - 3)$	$1/(2 * \|(x - 3))$
10	$sen(2x + 1)$	$2cos(2x + 1)$
11	$xsen(x) \wedge 2 + 3sen(2x)$	$sen(x) \wedge 2 + 2x \wedge 2cos(x) \wedge 2 + 6cos(2x)$
12	$sen(x \wedge 3) \wedge 3$	$9cos(x \wedge 3)x \wedge 2sen(x \wedge 3) \wedge 2$
13	$cos(3x \wedge 2 - x)$	$-sen(3x \wedge 2 - x)(6x - 1)$
14	$2cos(1 - x) \wedge 3$	$6sen(1 - x)cos(1 - x) \wedge 2$
15	$cos(x) + xcos(x \wedge 2) \wedge 2$	$-sen(x) - 4xsen(x \wedge 2)xcos(x \wedge 2) + cos(x \wedge 2) \wedge 2$
16	$sen(x)cos(x)$	$-sen(x) \wedge 2 + cos(x) \wedge 2$
17	$tg(1/(x + 3))$	$-(1 + tg(1/(x + 3)) \wedge 2)/((x + 3) \wedge 2)$
18	$tg(x \wedge 2 + 1) \wedge 2$	$4x(1 + tg(x \wedge 2 + 1) \wedge 2)tg(x \wedge 2 + 1)$
19	$cos(x) \wedge 2 + tg(xsen(x) \wedge 2)$	$-2sen(x)cos(x) + (1 + tg(xsen(x) \wedge 2) \wedge 2)(2xcos(x)sen(x) + sen(x) \wedge 2)$
20	$tg(\|(x \wedge 2 + 1))^2 + tg(cos(x))$	$2x/\|(x \wedge 2 + 1)$
21	$arcsen(x \wedge 2 + 1)$	$2x/\|(1 - (x \wedge 2 + 1) \wedge 2)$
22	$sen(x) \wedge 2 + arcsen(x \wedge 2)$	$2xsen(x) + (2x)/\|(1 - x \wedge 4)$
23	$2 + arcsen(cos(x) \wedge 2) \wedge 2$	$-(2sen(2x)arcsen(cos(x) \wedge 2))/\|(1 - cos(x) \wedge 4)$
24	$arcsen(1/x) + sen(1/x)$	$-1/(x * \|(x \wedge 2 - 1)) - cos(1/x)/(x \wedge 2)$
25	$arccos(x)/x$	$(-x - \|(1 - x \wedge 2)arcsen(cos(x) \wedge 2)/(x \wedge 2 * \|(1 - x \wedge 2))$
26	$sen(x)arccos(2x)$	$-2sen(x)/\|(1 - 4x \wedge 2) + cos(x)arccos(2x)$
27	$x \wedge 3 * arccos(\|(x \wedge 2 - 1))$	$3x \wedge 2arccos(\|(x \wedge 2 - 1)) - x^4/((\|x \wedge 2 - 1)(\|2 - x \wedge 2))$
28	$arctg(\|(x \wedge 2 + 1))$	$x/(\|(x \wedge 2 + 1)(x \wedge 2 + 2))$
29	$1 + arctg(x) \wedge 2$	$2arctg(x)/(1 + x \wedge 2)$
30	$e(-x/2)$	$-0.5e(-0.5x)$
31	$(x - 1) \wedge 2 * e(-x)$	$-((x - 1) \wedge 2)e(-x) + 2(x - 1)e(-x)$
32	$e(arcsen(x))$	$e(arcsen(x))/\|(1 - x \wedge 2)$
33	$e(x)sen(x) + e(1/x)$	$e(x)cos(x) + e(x)sen(x) - e(1/x) * 1/x \wedge 2$
34	$log(x \wedge 2 + 1)$	$2x/(x \wedge 2 + 1)$
35	$log(sen(x))$	$cos(x)/sen(x)$
36	$log(e(3x) + x \wedge 2)$	$(3e(3x) + 2x)/(e(3x) + x \wedge 2)$
37	$cos(log(arctg(x)))$	$-sen(log(arctg(x)))/((1 + x \wedge 2)arctg(x))$
38	$(5x) \wedge (log(x))$	$5log(x)(5x) \wedge (log(x) - 1) + (log(5x)(5x) \wedge log(x))/x$
39	$arctg(x) \wedge sen(x)$	$(sen(x)arctg(x) \wedge (sen(x) - 1))/(1 + x \wedge 2) + log(arctg(x))arctg(x) \wedge sen(x)cos(x)$

Tabela 6.31 Testes de composição de regras de derivação

Capítulo 7

Execução dos testes e análise dos resultados

7.1 Execução dos testes

Os testes apresentados no capítulos anterior foram organizados em suites de testes e escritos no dialecto HtestML. Cada ficheiro contem uma ou mais suites de testes, conforme indicado na tabela 7.1.

Suite	Testes	Ficheiro
1	Conversão de formatos: Valores numéricos (Tabela 6.10)	`ext2int_const.xml`
2	Conversão de formatos: Formato externo e formato interno (Tabela 6.11)	`ext2int_transf.xml`
3	Conversão de formatos: Validação sintáctica (Tabela 6.12)	`ext2int_vsint.xml`
4	Conversão de formatos: Formato interno para formato externo (Tabela 6.13)	`ext2int_transf.xml`
5	Conversão de formatos: Representação em árvore (Tabelas 6.14 e 6.15)	`reparv.xml`
6	Simplificação: Operações algébricas básicas (Tabela 6.16)	`simpl_op.xml`
7	Simplificação: Propriedades dos operadores algébricos (Tabelas 6.17 e 6.18)	`simpl_op.xml`
8	Simplificação: Funções (Tabelas 6.19 e 6.20)	`simpl_func.xml`
9	Simplificação: Validação semântica (Tabela 6.21)	`simpl_vsem.xml`
10	Simplificação: Qualidade (Tabelas 6.22, 6.23, 6.24, 6.25, 6.26, 6.27, 6.28 e 6.29)	`simpl_qualid.xml`
11	Derivação de expressões (Tabelas 6.30 e 6.31)	`derivar.xml`

Tabela 7.1 Execução de testes

O Testador é invocado da linha de comandos de um interpretador Haskell, como no exemplo seguinte:

```
Testador> testador
*** Teste de programas em Haskell ***
Ficheiro XML com os testes a efectuar: derivar
Executando testes em 'derivar.xml' ...
Resultados em 'derivar_2005June6-001.txt'
```

Segue-se um extracto do ficheiro de resultados da execução Testador com o ficheiro 'derivar.xml':

```
Resultados dos testes em 'derivar.xml'
Execução n° 004 de 2005June6

Casos: 65  Tentados: 0  Erros: 0  Falhas: 0
Casos: 65  Tentados: 1  Erros: 0  Falhas: 0
Casos: 65  Tentados: 2  Erros: 0  Falhas: 0
Casos: 65  Tentados: 3  Erros: 0  Falhas: 0
Casos: 65  Tentados: 4  Erros: 0  Falhas: 0
Casos: 65  Tentados: 5  Erros: 0  Falhas: 0

### Falha em: /TESTE[NOME='Derivacao de Expressoes']/TESTE[NOME='Derivadas simples']/
CASO_TESTE[FUNCAO='derivar' and AVALIAR='x^(x+1)']
esperado: "x^x(x+1)+log(x)x^(x+1)"
  obtido: "x^(x+1-1)(x+1)+log(x)x^(x+1)"

Casos: 65  Tentados: 6  Erros: 0  Falhas: 1
Casos: 65  Tentados: 7  Erros: 0  Falhas: 1
Casos: 65  Tentados: 8  Erros: 0  Falhas: 1
Casos: 65  Tentados: 9  Erros: 0  Falhas: 1
Casos: 65  Tentados: 10  Erros: 0  Falhas: 1
Casos: 65  Tentados: 11  Erros: 0  Falhas: 1
Casos: 65  Tentados: 12  Erros: 0  Falhas: 1

### Falha em: /TESTE[NOME='Derivacao de Expressoes']/TESTE[NOME='Derivadas simples']/
CASO_TESTE[FUNCAO='derivar' and AVALIAR='tg(3x^2)']
esperado: "6x(1+tg(3x^2)^2)"
  obtido: "6(1+tg(3x^2)^2)x"

Casos: 65  Tentados: 13  Erros: 0  Falhas: 2

### Falha em: /TESTE[NOME='Derivacao de Expressoes']/TESTE[NOME='Derivadas simples']/
CASO_TESTE[FUNCAO='derivar' and AVALIAR='arcsen(x)']
esperado: "1/|(1+x^2)"
  obtido: "1/((1+x^2)^0.5)"

Casos: 65  Tentados: 14  Erros: 0  Falhas: 3

### Falha em: /TESTE[NOME='Derivacao de Expressoes']/TESTE[NOME='Derivadas simples']/
CASO_TESTE[FUNCAO='derivar' and AVALIAR='arcsen(3x^2)']
esperado: "6x/|(1+9x^4)"
  obtido: "6x/((1+9x^4)^0.5)"

Casos: 65  Tentados: 15  Erros: 0  Falhas: 4

### Falha em: /TESTE[NOME='Derivacao de Expressoes']/TESTE[NOME='Derivadas simples']/
CASO_TESTE[FUNCAO='derivar' and AVALIAR='arccos(x)']
esperado: "-1/|(1-x^2)"
  obtido: "-1/(-(1+x^2)^0.5)"

Casos: 65  Tentados: 16  Erros: 0  Falhas: 5

### Falha em: /TESTE[NOME='Derivacao de Expressoes']/TESTE[NOME='Derivadas simples']/
CASO_TESTE[FUNCAO='derivar' and AVALIAR='arccos(3x^2)']
esperado: "-6x/|(1-9x^4)"
  obtido: "-6x/((1+9x^4)^0.5)"

Casos: 65  Tentados: 17  Erros: 0  Falhas: 6
Casos: 65  Tentados: 18  Erros: 0  Falhas: 6
Casos: 65  Tentados: 19  Erros: 0  Falhas: 6
...
```

7.2 Análise dos resultados

Cada execução de uma suite de testes produziu um ficheiro texto com os resultados. Antes de efectuar uma análise mais aprofundada aos resultados dos testes apresenta-se, na tabela 7.2, os valores finais dos contadores para cada suite:

Suite	Testes	Casos	Falhas	
1	Conversão de formatos: Valores numéricos	20	0	(0%)
2+4	Conversão de formatos: Formato externo e formato interno	12	1	(8%)
3	Conversão de formatos: Validação sintáctica	14	0	(0%)
5	Conversão de formatos: Representação em árvore	18	0	(0%)
6+7	Simplificação: Operações algébricas básicas + Propriedades	33	4	(12%)
8	Simplificação: Funções	19	1	(5%)
9	Simplificação: Validação semântica	4	1	(25%)
10	Simplificação: Qualidade	42	14	(33%)
11	Derivação de expressões	65	28	(43%)

Tabela 7.2 Resultados quantitativos da execução de testes

Os testes das suites 1 a 9 são testes que têm como objectivo a detecção de falhas na especificação da aplicação. Os testes da suites 10 visam medir a qualidade da simplificação efectuada. Os testes da suite 11 têm como objectivo simultaneamente detectar falhas no processo de derivação e medir a qualidade da simplificação que é efectuada à derivada.

No primeiro grupo (suites 1 a 9), verifica-se que o processo de simplificação a nível de operações básicas e suas propriedades é o que tem o nível de falhas mais elevado. A análise detalhada destas falhas permitirá averiguar sobre a sua gravidade.

No segundo grupo (suites 10 e 11), verifica-se que os níveis de falhas são igualmente elevados, o que revela que, de uma forma geral, a aplicação não simplifica totalmente as expressões, relativamente ao que seria esperado.

Nas secções seguintes é efectuada uma análise a estes resultados tendo em conta os objectivos definidos para cada teste.

7.2.1 Conversão de formatos

Os grupos de testes referentes a conversão de constantes do formato interno para o externo, validação sintáctica e representação em árvore não apresentaram falhas, pelo que se conclui que, no conjunto dos casos apresentados, estes processos são correctamente efectuados.

O grupo de testes que apresentou uma falha foi a conversão de expressões do formato interno para o formato externo. A falha é apresentada na tabela 7.3. Tal indica, no entanto, que a conversão de expressões do formato externo para o interno não teve falhas no âmbito dos casos de teste incluídos.

Avaliado	Esperado	Obtido
$2\|(x + -1 * (x \wedge 5 * cos(x)))$	$\|(x - 1(x \wedge 5cos(x)))$	$2\|(x - 1(x \wedge 5cos(x)))$

Tabela 7.3 Falhas da conversão de formato interno para formato externo

7.2.2 Simplificação

Operações básicas e as suas propriedades

O primeiro grupo de testes de simplificação incide na verificação das operações básicas e das suas propriedades. No âmbito destes testes foram detectadas 4 falhas em 33 casos de teste, representando 15%. A tabela 7.4 apresenta os casos de teste com falhas:

Avaliado	Esperado	Obtido
$5 + (-5)$	0	1
$2 + x + 3$	$x + 5$	$5 + x$
$(2 + x) + 3$	$x + 5$	$5 + x$
$2 + (x + 3)$	$x + 5$	$5 + x$

Tabela 7.4 Falhas da verificação de operações básicas e suas propriedades

Verifica-se que destas falhas há uma muito grave sendo as outras três menos graves. A primeira falha, na soma de um valor com o seu simétrico compromete seriamente todo o processo de simplificação, pelo deve corrigida de imediato. As três falhas restantes devem-se ao facto de ser esperado um polinómio ordenado por grau e a aplicação não efectuar essa ordenação. Esta falha vai manifestar-se em muitos dos testes seguintes em que são esperados polinómios ordenados por grau. Como não afecta a correcção do resultado, é considerada de menor gravidade.

Avaliação de funções

Este grupo de testes acusou um falha num conjunto de 19 casos, apresentada da tabela 7.5:

Avaliado	Esperado	Obtido
$cos(arcsen(1))$	0	$6.12303176911189e - 017$

Tabela 7.5 Falhas da conversão de avaliação de funções

Esta falha resulta da aproximação de valores utilizada pelo interpretador do Haskell. A intenção era obter o valor de π através da função $arcsen(1)$, no entanto, o valor obtido foi aproximado levando a que a função cos não tivesse o valor de π exacto, e consequentemente não assuma também exactamente o valor zero. O valor obtido está em notação científica podendo ser aproximado pelo valor zero. Daí se conclui que, sob ponto de vista de avaliação de funções, a falha é de pouca gravidade.

Validação Semântica

Este grupo de testes acusou uma falha na situação de divisão por zero. A falha não se verificou na validação em si mas na forma como o denominador foi simplificado, sendo o primeiro caso da tabela 7.4. Conclui-se que o processo de validação semântica não apresenta falhas da sua responsabilidade.

Qualidade da simplificação

Neste grupo de testes ocorreram 14 falhas nos 42 casos de teste, representando 33%, apresentadas na tabela 7.6:

Avaliado	**Esperado/Obtido**
$4 + 3x + x \wedge 2 + 5x \wedge 2 + 4x + 6$	$6x \wedge 2 + 7x + 10$
	$10 + 7x + 6x \wedge 2$
$(x \wedge 2 + 3x + 4) + 2(5x \wedge 2 + 4x + 6)$	$x \wedge 2 + 3x + 4 + 2(5x \wedge 2 + 4x + 6)$
	$x \wedge 2 + 3x + 4 + (2(5x \wedge 2 + 4x + 6))$
$4 + 3x + x \wedge 2$	$x \wedge 2 + 3x + 4$
	$4 + 3x + x \wedge 2$
$(x + 1) - (x + 1)$	0
	$x + 1 + (-(x + 1))$
$x \wedge 2 + 3x + 4 - 5x \wedge 2 - 4x - 6 + 4x \wedge 2 + x + 2$	0
	$0x \wedge 2 + 0x$
$x(x + 1)(x + 1)(x + 2)$	$(x + 1) \wedge 2x(x + 2)$
	$(x + 2)((x + 1) \wedge 2)x$
$(x + 1)x(x + 1)(x + 2)$	$(x + 1) \wedge 2x(x + 2)$
	$(x + 2)x((x + 1) \wedge 2)$
$(x + 1) \wedge 2(x + 2)(x + 1) \wedge 3$	$(x + 2)(x + 1) \wedge 5$
	$(x + 2)((x + 1) \wedge 5)$
$sen(x)cos(x)sen(x)$	$sen(x) \wedge 2cos(x)$
	$cos(x)sen(x) \wedge 2$
$sen(x)x * cos(x)sen(x) \wedge 2$	$sen(x) \wedge 3 * x * cos(x)$
	$cos(x)xsen(x) \wedge 3$
$x(x + 1)/x$	$x + 1$
	$(x + 1)x \wedge 0$
$(x + 1)(x + 2)/(x + 1)$	$x + 2$
	$(x + 2)((x + 1) \wedge 0)$
$((x + 1)(x + 2))/(x + 1)$	$x + 2$
	$((x + 1) \wedge 0)(x + 2)$
$x \wedge (x + 1) + x \wedge (x + 1)$	$2x \wedge (x + 1)$
	$2(x \wedge (x + 1))$

Tabela 7.6 Falhas na qualidade da simplificação de expressões

Tal como em grupos anteriores, as falhas variam em nível de gravidade. É possível identificar os problemas recorrentes que afectam a qualidade da simplificação:

- Nos polinómios, os monómios não são ordenados por grau;
- Em produtos e divisões, permanecem monómios com expoente zero que deviam ser simplificados para um e desaparecerem do produto;
- Em somas, permanecem factores que multiplicam por zero que deviam ser eliminados;
- Em diferenças de expressões iguais, há casos em que a expressão resultante é mais complicada do que a original;
- Verifica-se a presença de operadores de precedência redundantes na expressão;
- Em produtos de funções, a ordem dos factores é sempre invertida.

7.2.3 Derivação

Os testes de derivação englobam também testes de simplificação. Recorde-se que uma expressão antes de ser derivada é simplificada. A expressão resultante da derivada é também simplificada. Tendo-se verificado que o processo de simplificação tem algumas falhas importantes, é de esperar que os resultados dos testes de derivação apresentem muitas falhas. Por esse motivo, um sub-conjunto inicial desses testes analisa somente as regras de derivação sem simplificação. Os testes que se seguem a esse subconjunto incluem expressões que necessitam de simplificação e originam a derivadas que são também simplificadas.

Neste grupo de testes ocorreram 28 falhas nos 65 casos de teste, representando 43%. Verifica-se que na generalidade estas falhas são as mesmas analisada previamente no processo de simplificação, pelo que serão apenas apresentadas as que se devem exclusivamente ao processo de derivação ou que apresentam novas situações anómalas no processo de simplificação. Estas falhas são apresentadas na tabela 7.6:

Avaliado	**Esperado**	**Obtido**
$arcsen(x)$	$1/\|(1+x \wedge 2)$	$1/((1+x \wedge 2) \wedge 0.5)$
$arcsen(3x \wedge 2)$	$6x/\|(1+9x \wedge 4)$	$6x/((1+9x \wedge 4) \wedge 0.5)$
$arccos(x)$	$-1/\|(1-x \wedge 2)$	$-1/(-(1+x \wedge 2) \wedge 0.5)$
$arccos(3x \wedge 2)$	$-6x/\|(1-9x \wedge 4)$	$-6x/((1+9x \wedge 4) \wedge 0.5)$
$1/(x+3)$	$-1/(x+3) \wedge 2$	$-1/(-(x+3) \wedge 2)$

Tabela 7.7 Falhas no processo de derivação de expressões

Os dois primeiros casos apresentam falhas na simplificação. A raiz quadrada é avaliada como uma potência levando a discrepância nos resultados. Embora o resultado da derivação esteja correcto, não está apresentando da forma mais conveniente, que seria com o operador de radiciação. Os dois casos seguintes representam a única situação em que o cálculo da derivada é efectuado de forma errada. A regra de derivação da função *arccos* está implementada de forma diferente da especificação havendo no denominador uma troca do sinal + onde devia estar um sinal −. Tal leva à construção de uma expressão

que não corresponde à derivada da função. O último caso apresenta também um erro no cálculo. O denominador tem um sinal negativo que não era esperado no resultado.

Todas as outras falhas (não incluídas na tabela 7.7), devem-se exclusivamente a detalhes de simplificação implementados de forma diferente da esperada. Por esse motivo, as derivadas calculadas estão correctas, mas simplificadas de forma diferente ou incompleta, de acordo com as falhas já identificadas no processo de simplificação.

Capítulo 8

Conclusão

Este trabalho tinha vários objectivos que foram lançando desafios à medida que o desenvolvimento avançava. O primeiro deles foi a concepção de uma arquitectura de testes de programas em Haskell. Este objectivo foi atingido através da arquitectura de quatro componentes desenhada. A proposta apresentada neste trabalho constitui um ponto de partida passível de ser melhorado e implementado com mais funcionalidades. Para atingir este objectivo, foi necessário efectuar o estudo de uma *framework* de testes unitários. Sendo a linguagem Haskell o âmbito do ambiente de testes, a *framework* escolhida terá que ser também implementada nesta linguagem. Após alguma pesquisa na internet, foi encontrada a *framework* HUnit 1.0 que serviu de base à arquitectura.

Um segundo objectivo do trabalho foi a definição de uma estratégia de concepção de casos de teste. A estratégia que melhor se adequa à arquitectura desenvolvida é a *data driven* que consiste em executar sempre o mesmo código de teste, fazendo variar apenas a função a testar e o seu argumento. Para sistematizar e agilizar este processo, desenvolveu-se um dialecto XML, o qual de denominou de HtestML. Este dialecto, não tendo sido inicialmente definido como objectivo, revelou-se a representação mais adequada para a natureza hierárquica da organização de casos de teste do HUnit 1.0. A concepção de casos de teste obrigou à análise de regras de especificação. Os casos de teste foram concebidos de forma a cobrir essas regras de forma verificar o seu cumprimento. Paralelamente, dada a natureza dos processos de simplificação e derivação de expressões matemáticas, conceberam-se casos de testes destinados a medir a qualidade das expressões produzidas por estes processos. Esta concepção implicou separar os vários cenários de criação de expressões em classes de equivalência para que o número de testes fosse o mínimo necessário para cobrir as regras de especificação.

Como objectivo final, pretendeu-se detectar falhas na aplicação em teste. Foram detectadas algumas falhas, no entanto, reconhece-se que tal tarefa não é simples e exigiria um tempo de execução mais alargado para obter resultados mais completos. A detecção de falhas na aplicação é um exemplo meramente ilustrativo das capacidades da arquitectura de testes.

As perspectivas de trabalho futuro incluem o enriquecimento do dialecto HtestML, uma saída de resultados também para HtestML em vez de *plain text* e melhoramentos ao componente Testador de forma a incluir testes directos em funções com *output* para IO e tornar a sua execução mais rápida.

O desenvolvimento deste projecto revelou-se uma tarefa interessante tendo permitido tomar um primeiro contacto com uma ferramenta de teste de *software*.

Capítulo 9

Referências

Bibliografia:

- BURNSTEIN, Ilene - *Practical Software Testing*, Springer, 2003
- MITCHELL, John C. - *Concepts In Programming Languages*, Cambridge University Press, 2003
- THOMPSON, Simon - *Haskell, The Craft of Functional Programming*, Addison-Wesley, 1996
- ABELLANAS, Lorenzo, SPIEGEL, Murray R. - *Fórmulas e Tabelas de Matemática Aplicada*, McGraw-Hill, 1990

Sítios na *web*:

- http://www.haskell.org – Sítio oficial da comunidade de programadores em Haskell
- http://hunit.sourceforge.net – Sítio da *framework* HUnit 1.0