Hogebomen

Lisette de Schipper (s1396250) en Micky Faas (s1407937)

Inleiding

Al vanaf de middelbare school moeten we afgeleiden nemen van functies met een of meerdere onbekenden. Nu hebben wij een progrAmma geschreven dat dit werk van de gebruiker overneemt. De gebruiker vult simpelweg een expressie in prefix notatie en het programma doet vervolgens al het werk door middel van een expressieboom.

Werkwijze

De broncode van het programma bestaat uit de volgende bestanden:

- ExpressionAtom.cc
- ExoressionAtom.h
- ExpressionTree.cc
- ExpressionTree.h
- main.cc
- main2.cc
- Tree.h
- TreeNode.h
- TreeNodeIterator.h

In de terminal kun je in `hogebomen` "make" typen en vervolgens naar de `bin`-directory gaan, om daar `./hogebomen` te runnen.

class TreeNode

Hier staan alle knopen gedefinieerd. Van elke knoop leggen we het volgende vast:

- inhoud
- wie de ouder is, en wie de kinderen zijn

Daarnaast zijn er nog een aantal functies gedefinieerd, die allemaal voor zich spreken.

class Tree

De meeste functies hier spreken voor zich.
Als er een kind toegevoegd moet worden aan een knoop (`insert()`), maar die knoop is al vol, zijn er drie mogelijke reacties door het programma. Dit geldt ook voor het vervangen van een knoop (`replace()`). Dit gedrag kan als argument aan deze twee functies worden meegegeven.

- ABORT_ON_EXISTING, het programma wordt afgebroken en 0 wordt geretourneerd.
- MOVE_EXISTING, maak van de ouders kind een kind van de nieuwe knoop
- DELETE_EXISTING, verwijder een van de kinderen

Tot slot is er de functie `pushBack()` die een knoop toevoegt op de eerste volgende plek (gaat uit van een volle binaire boom, van links naar rechts gevuld).

class TreeNodeIterator

Dit bestand coördineert de drie mogelijke wandelingen door de boom: in-order wandeling, pre-order wandeling en post-order wandeling. In onze implementatie kun je erdoorheen lopen door gebruik te maken van een iterator. Een voorbeeld van het gebruik van zo'n iterator is als volgt:

```
Tree<char> tree;

//wat waardes om de boom te vullen
    tree.pushBack( 'a' );
    tree.pushBack( 'b' );
    tree.pushBack( 'c' );
    tree.pushBack( 'd' );

// in-orde wandeling
Tree<char>::iterator_in it( tree.begin_in( ) );

for( ; it !=tree.end_in( ); ++it ) {
    cout << *it << " ";
}

// of de (simpelere) standaard pre-orde
// dmv de begin() en end() functies (C++11)
for( auto c : tree )
    cout << c << endl;
```

De (abstracte) klasse (`TreeNodeIterator`) is een klasse die gebruikt wordt door de klassen die deze overerven/specialiseren. Deze klasse bevat dus een aantal algemene functies. Zo wordt hier de "++it" uit het voorbeeld hierboven gedefineerd, deze roept in de kind-klassen de virtuele functie (`next()`) aan om de volgende stap te zetten. De operatie "++" kunnen we dus ook gebruiken bij de andere 2 wandelingen.

De volgende 3 klassen `TreeNodeIterator_in`, `TreeNodeIterator_pre` en `TreeNodeIterator_post` bevatten allemaal de logica voor de bijbehorende wandelingen. Deze wandelingen zijn vrij simpel zonder stack geimplementeerd omdat we ervoor hebben gekozen om elke `TreeNode` een pointer naar zijn ouder te geven.

class ExpressionAtom

In ExpressionAtom staan alle inhouden/typen van knopen gedefinieerd die er gebruikt kunnen worden en ook een aantal operaties die we op ze kunnen uitvoeren. Zo worden ==, +, −, ∗, /, <,>,<= en >= ge-overload zodat ze door deze atomen gebruikt kunnen worden.
De types die door ExpressionAtom gebruikt kunnen worden:

- Integer
- Float/Kommagetal
- Breuk (aparte struct `Fraction`)
- Variabele
- Operator
  - +
  - -
  - *
  - /
  - ^
- Functie
  - sin
  - cos
  - tan
  - log
  - ln
  - wortel
  - absolute waardes
  - e
  - pi
  - unaire -

class ExpressionTree

Om strings te ontleden naar bomen maken we gebruik van een tokenizer (`tokenize()`) en een parser (`fromString()`). De tokenizer zet elk element om in een token van type `ExpressionAtom`. Al die tokens zijn de input van de parser die ze verder verwerkt tot een boom. De uitkomst is altijd ondubbelzinnig omdat op voorhand de ariteit van elke token bekend is (dmv van `ExpressionAtom::arity()`). De `differentiate()`, `generateInOrder)(` en `simplify()`-functies pakken hun problemen recursief aan.

Bij het evalueren worden alle variabelen door conrete waarden vervangen door de functies `mapVariables()` en/of `mapVariable()`. Het resultaat wordt hierna versimpeld in `simplify()`

hogebomen

Simpele interface voor de gebruiker om te differentiëren, evalueren, simplificeren en converteren naar dot-notatie. Bronbestand `main.cc` wordt gebouwd naar `bin/hogebomen`

hogebomen2

Klein testprogramma. Als je deze code uitvoerd, zul je zien dat de 3 wandelingen (pre, post en in) goed uitgevoerd worden. Bronbestand `main2.cc` wordt gebouwd naar `bin/hogebomen2`

Voorbeelden

Differentiëren

Differentieerbare functies:

- $tan(f(x))$
- $sin(f(x))$
- $cos(f(x))$
- $ln(f(x))$
- $sqrt(f(x))$
- $log(f(x))$
- $abs(f(x))$
- $f(x) + g(x)$
- $f(x) - g(x)$
- $f(x)/g(x)$
- $f(x) * g(x)$
- $f(x)^{g(x)}$

- x^n
- $x^{f(x)}$
- $e^{f(x)}$
- $n^{f(x)}$

en combinaties hiervan.

Nu een aantal voorbeelden. De expressies zijn gegeven in de infix notatie en hun afgeleides ook. Alle afgeleides zijn correct.

tan(ax)

Oorspronkelijke expressie: $tan(x^-7)$
Uitkomst: $((cos(x^{-7}) * ((-7/(x^8)) * cos(x^{-7}))) - (sin(x^{-7}) * ((-(-7/(x^8))) * sin(x^{-7}))))/(cos(x^{-7})^2)$

sin(ax)

Oorspronkelijke expressie: $sin(x * 4)$
Uitkomst: $4cos(x * 4)$

cos(ax)

Oorspronkelijke expressie: $cos(x/ * t^3x)$
Uitkomst: $(-((((t^3) * x) - (x * ((x*) + (t^3))))/(((t^3) * x)^2))) * sin(x/((t^3) * x))$

ln(ax)

Oorspronkelijke expressie: $ln(4 * x + x)$
Uitkomst: $5/((4x) + x)$

sqrt(ax)

Oorspronkelijke expressie: $sqrt(ln(12.3 * x^2))$
Uitkomst: $((12.3(2x))/(12.3(x^2)))/(2sqrtln(12.3(x^2)))$

log(ax)

Oorspronkelijke expressie: ${}^4log(5x)$
Uitkomst: $(1/ln(4)) * (5/(5x))$

abs(ax)

Oorspronkelijke expressie: $abs(cos(3 * x))$
Uitkomst: $(cos(3x) * (-3sin(3x)))/abscos(3x)$

f(x) + g(x)

Oorspronkelijke expressie: $ln(4x) + (4/(x^2))$
Uitkomst: $(4/(4x)) + ((0 - (4(2x)))/((x^2)^2))$

f(x) / g(x)

Oorspronkelijke expressie: $3x/(2^x)$
Uitkomst: $(((2^x) * 3) - ((3x) * (ln(2) * (2^x))))/((2^x)^2)$

f(x) * g(x)

Oorspronkelijke expressie: $x * 3$
Uitkomst: 3

f(x) ^ g(x)

Oorspronkelijke expressie: x^x
Uitkomst: $((x * (1/x)) + ln(x)) * (e^{(ln(x)*x)})$

Wandelingen

main2.cc bevat de volgende boom:

```
          =
    *              /
+     -        :      %
1 2   3 4      5 6    7 8
```

Output van main2.cc:

in-order traversal: 1 + 2 * 3 - 4 = 5 : 6 / 7 % 8
post-order traversal: 1 2 + 3 4 - * 5 6 : 7 8 % / =
pre-order traversal: = * + 1 2 - 3 4 / : 5 6 % 7 8

Appendix

ExpressionAtom.h

```
1  /**
2   * ExpressionAtom:
3   *
4   * @author  Micky Faas (s1407937)
5   * @author  Lisette de Schipper (s1396250)
6   * @file    ExpressionAtom.h
7   * @date    26-10-2014
8   **/
9
10 #ifndef EXPRESSIONATOM_H
11 #define EXPRESSIONATOM_H
12
13 #include <ostream>
14 #include <string>
15 #include <cmath>
16
17 typedef struct {
18     int numerator;
```

```
19      int denominator;
20  } Fraction;
21
22  /**
23  * @function  operator==( )
24  * @abstract  Test equality for two Fractions
25  * @param     lhs and rhs are two sides of the comparison
26  * @return    true upon equality
27  * @post      Two Fraction are equal if
28  *            lhs.numerator/lhs.denominator == rhs.numerator/rhs.denominator
29  **/
30  bool operator ==( const Fraction& lhs, const Fraction& rhs );
31
32  /**
33  * @function  Arithmetic operators +, -, *, /
34  * @abstract  Arithmetic result of two Fractions
35  * @param     lhs and rhs are two sides of the expression
36  **/
37  Fraction operator+( const Fraction& lhs, const Fraction& rhs );
38  Fraction operator-( const Fraction& lhs, const Fraction& rhs );
39  Fraction operator*( const Fraction& lhs, const Fraction& rhs );
40  Fraction operator/( const Fraction& lhs, const Fraction& rhs );
41
42  using namespace std;
43
44  class ExpressionAtom {
45      public:
46          enum AtomType {
47              UNDEFINED =0x0,
48              INTEGER_OPERAND,
49              FLOAT_OPERAND,
50              FRACTION_OPERAND,
51              NAMED_OPERAND, // Variable
52              OPERATOR,
53              FUNCTION
54          };
55
56          enum OperatorType {
57              SUM,
58              DIFFERENCE,
59              PRODUCT,
60              DIVISION,
61              EXPONENT
62          };
63
64          enum Function {
65              SIN,
66              COS,
67              TAN,
68              LOG,
69              LN,
70              SQRT,
71              ABS,
72              E,
```

```
73              PI,
74              UNARY_MINUS
75          };
76
77          /**
78           * @function  ExpressionAtom( )
79           * @abstract  Constructor, defines an ExpressionAtom for various types
80           * @param     Either one of AtomType, OperatorType, Function,
81           *            float, long int, Fraction or string
82           * @post      ExpressionAtom is always valid, containing the
83           *            supplied value. No argument yields UNDEFINED.
84           **/
85           ExpressionAtom( AtomType t =UNDEFINED, long int atom =0l );
86           ExpressionAtom( float atom );
87           ExpressionAtom( long int atom );
88           ExpressionAtom( string var );
89           ExpressionAtom( OperatorType op );
90           ExpressionAtom( Function func );
91           ExpressionAtom( Fraction frac );
92
93          /**
94           * @function  operator==( )
95           * @abstract  Test equality for two ExpressionAtom
96           * @param     ExpressionAtom or either one of AtomType, OperatorType,
97           *            Function, float, long int, Fraction  or string
98           * @return    true upon equality
99           * @post      Two ExpressionAtoms are equal if
100          *            - their types are equal
101          *            - their value is equal
102          *            - they are not UNDEFINED
103          **/
104          bool operator ==( const ExpressionAtom& rhs ) const;
105
106         /**
107          * @function  Inquality operators <, >, <= and >=
108          * @abstract  Test equality for two ExpressionAtoms
109          * @param     ExpressionAtom or either one of AtomType, OperatorType,
110          *            Function, float, long int, Fraction  or string
111          * @return    true upon resp. lt, gt, lte or gte
112          * @pre       Both operands should be of the numeric operand type
113          *            Types do not have to be equal
114          * @post      always false if !isNumericOperand( ) or UNDEFINED
115          **/
116          bool operator <( const ExpressionAtom& rhs ) const;
117          bool operator >( const ExpressionAtom& rhs ) const;
118          bool operator <=( const ExpressionAtom& rhs ) const;
119          bool operator >=( const ExpressionAtom& rhs ) const;
120
121         /**
122          * @function  Arithmetic operators +, -, *, /
123          * @abstract  Arithmetic result of two ExpressionAtoms
124          * @param     ExpressionAtom or either one of AtomType, OperatorType,
125          *            Function, float, long int, Fraction  or string
126          * @return    ExpressionAtom (xvalue) containing the result
```

```
127          *            The type of this ExpressionAtom doesn't need to be
128          *            equal to one of the operand's types
129          * @pre       Both operands should be of the numeric operand type
130          *            Types do not have to be equal
131          * @post      undefined if !isNumericOperand( ) or UNDEFINED
132          **/
133          ExpressionAtom operator+( const ExpressionAtom& rhs ) const;
134          ExpressionAtom operator-( const ExpressionAtom& rhs ) const;
135          ExpressionAtom operator*( const ExpressionAtom& rhs ) const;
136          ExpressionAtom operator/( const ExpressionAtom& rhs ) const;
137
138         /**
139          * @function  pow( )
140          * @abstract  Raise to power
141          * @param     ExpressionAtom or Either one of AtomType, OperatorType,
142          *            Function, float, long int, Fraction  or string
143          * @return    ExpressionAtom (xvalue) containing the result
144          *            The type of this ExpressionAtom doesn't need to be
145          *            equal to one of the operand's types
146          * @pre       Both operands should be of the numeric operand type
147          *            Types do not have to be equal
148          * @post      undefined if !isNumericOperand( ) or UNDEFINED
149          **/
150          ExpressionAtom pow( const ExpressionAtom& power ) const;
151         /**
152          * @function  sqrt( )
153          * @abstract  Square root
154          * @pre       Instance should be of the numeric operand type
155          *            Types do not have to be equal
156          * @return    ExpressionAtom (xvalue) containing the result
157          *            The type of this ExpressionAtom doesn't need to be
158          *            equal to the operand's types
159          * @post      undefined if !isNumericOperand( ) or UNDEFINED
160          **/
161          ExpressionAtom sqrt( ) const;
162
163         /**
164          * @function  setters
165          * @abstract  sets ExpressionAtom to a given value
166          * @param     Either one of AtomType, OperatorType,
167          *            Function, float, long int, Fraction  or string
168          * @post      The type is changed to match the new value
169          **/
170          void setFloat( float d )
171              { m_type =FLOAT_OPERAND; m_atom.float_atom =std::move( d ); }
172          void setInteger( long int i )
173              { m_type =INTEGER_OPERAND; m_atom.integer_atom =std::move( i ); }
174          void setFraction( const Fraction& frac )
175              { m_type =FRACTION_OPERAND; m_atom.fraction_atom =std::move( frac ); }
176          void setFunction( Function f )
177              { m_type =FUNCTION; m_atom.integer_atom =std::move( f ); }
178          void setOperator( OperatorType op )
179              { m_type =OPERATOR; m_atom.integer_atom =std::move( op ); }
180          void setNamed( string str )
```

```
181             { m_type =NAMED_OPERAND; m_named_atom =std::move( str ); }
182
183         /**
184          * @function  getters
185          * @abstract  Return the value as a certain type
186          * @return    Returns the value as the requested type
187          * @pre       Type should match the requested datatype
188          * @post      undefined if type doesn't match or UNDEFINED
189          **/
190          float getFloat( ) const { return m_atom.float_atom; }
191          long int getInteger( ) const { return m_atom.integer_atom; }
192          Fraction getFraction( ) const { return m_atom.fraction_atom; }
193          int getFunction( ) const { return (int)m_atom.integer_atom; }
194          int getOperator( ) const { return (int)m_atom.integer_atom; }
195          string getNamed( ) const { return m_named_atom; }
196
197         /**
198          * @function  isNumericOperand( )
199          * @abstract  Returns whether this instance holds a numeric type
200          * @return    bool with the result
201          **/
202          bool isNumericOperand( ) const {
203              return m_type == FLOAT_OPERAND
204                  || m_type == INTEGER_OPERAND
205                  || m_type == FRACTION_OPERAND; }
206
207         /**
208          * @function  numeric casting functions
209          * @abstract  Casts the value to a certain type
210          * @return    Returns the value as the requested type
211          * @pre       Type should be a numeric operand
212          *            toFloat( ) and toInteger( ) are defined for
213          *            FLOAT_OPERAND, INTEGER_OPERAND and FRACTION_OPERAND
214          *            toFraction( ) is defined for INTEGER_OPERAND and FRACTION
215          * @post      undefined if !isNumericOperand( )
216          **/
217          float toFloat( ) const;
218          long int toInteger( ) const;
219          Fraction toFraction( ) const;
220
221         /**
222          * @function  type( )
223          * @abstract  Gives the specified type
224          * @return    One of AtomType
225          **/
226          AtomType type( ) const { return m_type; }
227
228         /**
229          * @function  arity( )
230          * @abstract  Returns the arity of the specified type
231          * @return    Arity ranging from 0 to 2
232          **/
233          short arity( ) const;
234
```

```
235     private:
236         union {
237             long int integer_atom;
238             float float_atom;
239             Fraction fraction_atom;
240         } m_atom;
241         string m_named_atom;
242         AtomType m_type;
243 };
244 
245 /**
246 * @function  operator <<( ostream& out, const ExpressionAtom& atom )
247 * @abstract  Overloads operator<< to support ExpressionAtom
248 * @return    an ostream with the contents of atom inserted
249 **/
250 ostream& operator <<( ostream& out, const ExpressionAtom& atom );
251 
252 #endif
```

ExpressionAtom.cc

```
1  /**
2   * ExpressionAtom:
3   *
4   * @author  Micky Faas (s1407937)
5   * @author  Lisette de Schipper (s1396250)
6   * @file    ExpressionAtom.cc
7   * @date    26-10-2014
8   **/
9 
10 #include "ExpressionAtom.h"
11 #include "ExpressionTree.h"
12 
13 /* Fraction overloads */
14 
15 bool operator ==( const Fraction& lhs, const Fraction& rhs ) {
16     // This function should be in general namespace
17     return ExpressionTree::compare( (float)lhs.numerator/(float)lhs.denominator,
18                                     (float)rhs.numerator/(float)rhs.denominator );
19 }
20 
21 Fraction operator+( const Fraction& lhs, const Fraction& rhs ) {
22     Fraction f;
23     if( lhs.denominator == rhs.denominator ) {
24         f.denominator =lhs.denominator;
25         f.numerator =lhs.numerator + rhs.numerator;
26     } else {
27         f.denominator =lhs.denominator * rhs.denominator;
28         f.numerator =lhs.numerator * rhs.denominator
29                     + rhs.numerator * lhs.denominator;
30     }
31     return f;
32 }
33 
```

```
34 Fraction operator-( const Fraction& lhs, const Fraction& rhs ) {
35     Fraction f;
36     if( lhs.denominator == rhs.denominator ) {
37         f.denominator =lhs.denominator;
38         f.numerator =lhs.numerator - rhs.numerator;
39     } else {
40         f.denominator =lhs.denominator * rhs.denominator;
41         f.numerator =lhs.numerator * rhs.denominator
42                     - rhs.numerator * lhs.denominator;
43     }
44     return f;
45 }
46
47 Fraction operator*( const Fraction& lhs, const Fraction& rhs ) {
48     Fraction f;
49     f.denominator =lhs.denominator * rhs.denominator;
50     f.numerator =lhs.numerator * rhs.numerator;
51     return f;
52 }
53
54 Fraction operator/( const Fraction& lhs, const Fraction& rhs ) {
55     Fraction f;
56     f.denominator =lhs.denominator * rhs.numerator;
57     f.numerator =lhs.numerator * rhs.denominator;
58     return f;
59 }
60
61 /* ExpressionAtom implementation */
62
63 ExpressionAtom::ExpressionAtom( AtomType t, long int atom ) : m_type( t ) {
64     m_atom.integer_atom =std::move( atom );
65 }
66
67 ExpressionAtom::ExpressionAtom( float atom ) : m_type( FLOAT_OPERAND ) {
68     m_atom.float_atom =std::move( atom );
69 }
70
71 ExpressionAtom::ExpressionAtom( long int atom ) : m_type( INTEGER_OPERAND ) {
72     m_atom.integer_atom =std::move( atom );
73 }
74
75 ExpressionAtom::ExpressionAtom( string var ) : m_type( NAMED_OPERAND ) {
76     m_named_atom =std::move( var );
77 }
78
79 ExpressionAtom::ExpressionAtom( OperatorType op ) : m_type( OPERATOR ) {
80     m_atom.integer_atom =std::move( op );
81 }
82
83 ExpressionAtom::ExpressionAtom( Function func ) : m_type( FUNCTION ) {
84     m_atom.integer_atom =std::move( func );
85 }
86
87 ExpressionAtom::ExpressionAtom( Fraction frac ) : m_type( FRACTION_OPERAND ) {
```

```
88      m_atom.fraction_atom =std::move( frac );
89  }
90
91  bool ExpressionAtom::operator ==( const ExpressionAtom& rhs ) const {
92      if( rhs.m_type != m_type )
93          return false;
94      switch( m_type ) {
95          case UNDEFINED:
96              return false;
97          case INTEGER_OPERAND:
98          case OPERATOR:
99          case FUNCTION:
100             return m_atom.integer_atom == rhs.m_atom.integer_atom;
101
102         case FLOAT_OPERAND:
103             return m_atom.float_atom == rhs.m_atom.float_atom;
104
105         case FRACTION_OPERAND:
106             return m_atom.fraction_atom == rhs.m_atom.fraction_atom;
107
108         case NAMED_OPERAND:
109             return m_named_atom == rhs.m_named_atom;
110
111     }
112     return false;
113 }
114
115 bool ExpressionAtom::operator <( const ExpressionAtom& rhs ) const {
116     if( !rhs.isNumericOperand( ) || !isNumericOperand( ) )
117         return false;
118     return toFloat( ) < rhs.toFloat( );
119 }
120
121 bool ExpressionAtom::operator >( const ExpressionAtom& rhs ) const {
122     if( !rhs.isNumericOperand( ) || !isNumericOperand( ) )
123         return false;
124     return toFloat( ) > rhs.toFloat( );
125 }
126
127 bool ExpressionAtom::operator <=( const ExpressionAtom& rhs ) const {
128     if( !rhs.isNumericOperand( ) || !isNumericOperand( ) )
129         return false;
130     return toFloat( ) <= rhs.toFloat( );
131 }
132
133 bool ExpressionAtom::operator >=( const ExpressionAtom& rhs ) const {
134     if( !rhs.isNumericOperand( ) || !isNumericOperand( ) )
135         return false;
136     return toFloat( ) >= rhs.toFloat( );
137 }
138
139 ExpressionAtom ExpressionAtom::operator+( const ExpressionAtom& rhs ) const {
140     ExpressionAtom a;
141     if( isNumericOperand( ) && rhs.isNumericOperand( ) ) {
```

```
142         if( m_type == FLOAT_OPERAND || rhs.m_type == FLOAT_OPERAND )
143             a.setFloat( toFloat( ) + rhs.toFloat( ) );
144         else if( m_type == FRACTION_OPERAND || rhs.m_type == FRACTION_OPERAND )
145             a.setFraction( toFraction( ) + rhs.toFraction( ) );
146         else
147             a.setInteger( getInteger( ) + rhs.getInteger( ) );
148     }
149     return a;
150 }
151
152 ExpressionAtom ExpressionAtom::operator-( const ExpressionAtom& rhs ) const {
153     ExpressionAtom a;
154     if( isNumericOperand( ) && rhs.isNumericOperand( ) ) {
155         if( m_type == FLOAT_OPERAND || rhs.m_type == FLOAT_OPERAND )
156             a.setFloat( toFloat( ) - rhs.toFloat( ) );
157         else if( m_type == FRACTION_OPERAND || rhs.m_type == FRACTION_OPERAND )
158             a.setFraction( toFraction( ) - rhs.toFraction( ) );
159         else
160             a.setInteger( getInteger( ) - rhs.getInteger( ) );
161     }
162     return a;
163 }
164
165 ExpressionAtom ExpressionAtom::operator*( const ExpressionAtom& rhs ) const {
166     ExpressionAtom a;
167     if( isNumericOperand( ) && rhs.isNumericOperand( ) ) {
168         if( m_type == FLOAT_OPERAND || rhs.m_type == FLOAT_OPERAND )
169             a.setFloat( toFloat( ) * rhs.toFloat( ) );
170         else if( m_type == FRACTION_OPERAND || rhs.m_type == FRACTION_OPERAND )
171             a.setFraction( toFraction( ) * rhs.toFraction( ) );
172         else
173             a.setInteger( getInteger( ) * rhs.getInteger( ) );
174     }
175     return a;
176 }
177
178 ExpressionAtom ExpressionAtom::operator/( const ExpressionAtom& rhs ) const {
179     ExpressionAtom a;
180     if( isNumericOperand( ) && rhs.isNumericOperand( ) ) {
181         if( m_type == FLOAT_OPERAND || rhs.m_type == FLOAT_OPERAND )
182             a.setFloat( toFloat( ) / rhs.toFloat( ) );
183         else if( m_type == FRACTION_OPERAND || rhs.m_type == FRACTION_OPERAND )
184             a.setFraction( toFraction( ) / rhs.toFraction( ) );
185         else
186             a.setInteger( getInteger( ) / rhs.getInteger( ) );
187     }
188     return a;
189 }
190
191 ExpressionAtom ExpressionAtom::pow( const ExpressionAtom& power ) const {
192     ExpressionAtom a;
193     if( isNumericOperand( ) && power.isNumericOperand( ) ) {
194
195         if( power.m_type == FRACTION_OPERAND
```

```
196             && power.m_atom.fraction_atom == Fraction( { 1, 2 } ) ) {
197             return sqrt( );
198         }
199         else if( m_type == FLOAT_OPERAND
200             || power.m_type == FLOAT_OPERAND
201             || power.m_type == FRACTION_OPERAND )
202             a.setFloat( ::powf( toFloat( ), power.toFloat( ) ) );
203         else if( m_type == FRACTION_OPERAND ) {
204             Fraction f;
205             f.numerator =m_atom.fraction_atom.numerator;
206             f.denominator =::pow( m_atom.fraction_atom.denominator,
207                                   power.getInteger( ) );
208             a.setFraction( f );
209         }
210         else {
211             if( power.getInteger( ) > 0 )
212                 a.setInteger( ::powl( getInteger( ), power.getInteger( ) ) );
213             else if( power.getInteger( ) == 0 )
214                 a.setInteger( 1 );
215             else {
216                 Fraction f;
217                 f.numerator =1;
218                 f.denominator =::pow( m_atom.integer_atom,
219                                       abs( power.m_atom.integer_atom ) );
220                 a.setFraction( f );
221             }
222         }
223     }
224     return a;
225 }
226
227 ExpressionAtom ExpressionAtom::sqrt( ) const {
228     ExpressionAtom a;
229     if( isNumericOperand( ) ) {
230         if( m_type == FLOAT_OPERAND  ) {
231             a.setFloat( ::sqrtf( toFloat( ) ) );
232         }
233         else if( m_type == FRACTION_OPERAND ) {
234             float f =::sqrtf( (float)m_atom.fraction_atom.denominator );
235             if( ceil( f ) == floor( f ) )
236                 a.setFraction(
237                     Fraction( { m_atom.fraction_atom.numerator, (int)f } ) );
238             else
239                 a.setFloat( f );
240         }
241         else {
242             float f =::sqrtf( (float)m_atom.integer_atom );
243             if( ceil( f ) == floor( f ) )
244                 a.setInteger( (int)f );
245             else
246                 a.setFloat( f );
247         }
248     }
249     return a;
```

```
250 }
251
252 float ExpressionAtom::toFloat( ) const {
253     if( m_type == INTEGER_OPERAND )
254         return (float)m_atom.integer_atom;
255     else if( m_type == FLOAT_OPERAND )
256         return m_atom.float_atom;
257     else if( m_type == FRACTION_OPERAND )
258         return (float)m_atom.fraction_atom.numerator /
259                (float)m_atom.fraction_atom.denominator;
260
261     return float( );
262 }
263 long int ExpressionAtom::toInteger( ) const {
264     if( m_type == INTEGER_OPERAND )
265         return m_atom.integer_atom;
266     else if( m_type == FLOAT_OPERAND )
267         return (long int)m_atom.float_atom;
268     else if( m_type == FRACTION_OPERAND )
269         return m_atom.fraction_atom.numerator /
270                m_atom.fraction_atom.denominator;
271
272     return int( );
273 }
274 Fraction ExpressionAtom::toFraction( ) const {
275     Fraction frac;
276     if( m_type == FRACTION_OPERAND )
277         return m_atom.fraction_atom;
278     else if( m_type == INTEGER_OPERAND ) {
279         frac.numerator =m_atom.integer_atom;
280         frac.denominator =1;
281     }
282     return frac;
283 }
284
285 short ExpressionAtom::arity( ) const {
286     switch( type( ) ) {
287         case ExpressionAtom::INTEGER_OPERAND:
288         case ExpressionAtom::FLOAT_OPERAND:
289         case ExpressionAtom::FRACTION_OPERAND:
290         case ExpressionAtom::NAMED_OPERAND:
291             return 0;
292
293         case ExpressionAtom::OPERATOR:
294             return 2;
295         case ExpressionAtom::FUNCTION:
296             switch( getFunction( ) ) {
297                 case ExpressionAtom::SIN:
298                 case ExpressionAtom::COS:
299                 case ExpressionAtom::TAN:
300                 case ExpressionAtom::LN:
301                 case ExpressionAtom::SQRT:
302                 case ExpressionAtom::ABS:
303                 case ExpressionAtom::UNARY_MINUS:
```

```
304                     return 1;
305                 case ExpressionAtom::E:
306                 case ExpressionAtom::PI:
307                     return 0;
308                 case ExpressionAtom::LOG:
309                     return 2;
310             }
311             break;
312         case ExpressionAtom::UNDEFINED:
313         default:
314             return 0;
315     }
316     return 0;
317 }
318
319 /* General namespace */
320
321 ostream& operator <<( ostream& out, const ExpressionAtom& atom ) {
322     switch( atom.type( ) ) {
323         case ExpressionAtom::INTEGER_OPERAND:
324             out << atom.getInteger( );
325             break;
326         case ExpressionAtom::FLOAT_OPERAND:
327             out << atom.getFloat( );
328             break;
329         case ExpressionAtom::FRACTION_OPERAND:
330             out << atom.getFraction( ).numerator << "/"
331                  << atom.getFraction( ).denominator;
332             break;
333         case ExpressionAtom::NAMED_OPERAND:
334             out << atom.getNamed( );
335             break;
336         case ExpressionAtom::OPERATOR:
337             switch( atom.getOperator( ) ) {
338                 case ExpressionAtom::SUM:
339                     out << "+";
340                     break;
341                 case ExpressionAtom::DIFFERENCE:
342                     out << "-";
343                     break;
344                 case ExpressionAtom::PRODUCT:
345                     out << "*";
346                     break;
347                 case ExpressionAtom::DIVISION:
348                     out << "/";
349                     break;
350                 case ExpressionAtom::EXPONENT:
351                     out << "^";
352                     break;
353             }
354             break;
355         case ExpressionAtom::FUNCTION:
356             switch( atom.getFunction( ) ) {
357                 case ExpressionAtom::SIN:
```

```
358                     out << "sin";
359                     break;
360                 case ExpressionAtom::COS:
361                     out << "cos";
362                     break;
363                 case ExpressionAtom::TAN:
364                     out << "tan";
365                     break;
366                 case ExpressionAtom::LOG:
367                     out << "log";
368                     break;
369                 case ExpressionAtom::LN:
370                     out << "ln";
371                     break;
372                 case ExpressionAtom::SQRT:
373                     out << "sqrt";
374                     break;
375                 case ExpressionAtom::ABS:
376                     out << "abs";
377                     break;
378                 case ExpressionAtom::E:
379                     out << "e";
380                     break;
381                 case ExpressionAtom::PI:
382                     out << "pi";
383                     break;
384                 case ExpressionAtom::UNARY_MINUS:
385                     out << "-";
386                     break;
387             }
388             break;
389         case ExpressionAtom::UNDEFINED:
390         default:
391             break;
392     }
393     return out;
394 }
```

ExpressionTree.h

```
1  /**
2   * ExpressionTree:
3   *
4   * @author Micky Faas (s1407937)
5   * @author Lisette de Schipper (s1396250)
6   * @file ExpressionTree.h
7   * @date 10-10-2014
8   **/
9
10 #ifndef EXPRESSIONTREE_H
11 #define EXPRESSIONTREE_H
12
13 #include "Tree.h"
14 #include "ExpressionAtom.h"
```

```
15 #include <fstream>
16 #include <string>
17 #include <exception>
18 #include <stdexcept>
19 #include <sstream>
20 #include <cmath>
21 #include <map>
22
23 using namespace std;
24
25 class ParserException : public exception
26 {
27     public:
28        ParserException( const string &str ) : s( str ) {}
29        ~ParserException() throw () {}
30        const char* what() const throw() { return s.c_str(); }
31
32     private:
33        string s;
34 };
35
36 class ExpressionTree : public Tree<ExpressionAtom>
37 {
38     public:
39        /**
40         * @function  ExpressionTree( )
41         * @abstract  Constructor, creates an object of the tree.
42         * @post      The tree has been declared.
43         **/
44         ExpressionTree( ) : Tree<ExpressionAtom>() { }
45
46        /**
47         * @function  ExpressionTree( )
48         * @abstract  fromString is called to make a tree from the string.
49         * @param     str, a string that will be parsed to create the three.
50         * @post      The tree has been declared and initialized.
51         **/
52         ExpressionTree( const string& str ) : Tree<ExpressionAtom>() {
53             fromString( str );
54         }
55
56        /**
57         * @function  tokenize( )
58         * @abstract  Breaks the string provided by fromString up into tokens
59         * @param     str, a string expression
60         * @return    tokenlist, a list of ExpressionAtom's
61         * @pre       str needs to be a correct space-separated string
62         * @post      We have tokens of the string
63         **/
64         static list<ExpressionAtom> tokenize( const string& str );
65
66        /**
67         * @function  fromString( )
68         * @abstract  calls tokenize to generate tokens from an expression and
```

```
69          *            fills the ExpressionTree with them.
70          * @param     expression, a string expression
71          * @post      The provided expression will be converted to an
72          *            ExpressionTree if it has the right syntax.
73          **/
74          void fromString( const string& expression );
75
76         /**
77          * @function  differentiate( )
78          * @abstract  calls the other differentiate function and returns the
79          *            derivative in the form of a tree
80          * @param     string varName, the variable
81          * @return    the derivative of the original function in the form of a
82          *            tree
83          * @pre       There needs to be a tree
84          * @post      Derivatree has been changed by the private differentiate
85          *            function.
86          **/
87          ExpressionTree differentiate( string varName );
88
89         /**
90          * @function  simplify( )
91          * @abstract  Performs mathematical simplification on the expression
92          * @post      Upon simplification, nodes may be deleted.
93          *            references and iterators may become invalid
94          **/
95          void simplify( );
96
97         /**
98          * @function  evaluate( )
99          * @abstract  Evaluates the tree as far as possible given a variable and
100         *            its mapping
101         * @return    A new ExpressionTree containing the evaluation (may be a
102         *            single node)
103         * @param     varName, variable name to match (e.g, 'x')
104         * @param     expr, expression to put in place of varName
105         **/
106         ExpressionTree evaluate( string varName, ExpressionAtom expr ) const;
107
108        /**
109         * @function  evaluate( )
110         * @abstract  Evaluates the tree as far as possible using a given mapping
111         * @return    A new ExpressionTree containing the evaluation (may be a
112         *            single node)
113         * @param     varmap, list of varName/expr pairs
114         **/
115         ExpressionTree evaluate( const map<string,ExpressionAtom>& varmap ) const;
116
117        /**
118         * @function  mapVariable( )
119         * @abstract  Replaces a variable by an expression
120         * @param     varName, variable name to match (e.g, 'x')
121         * @param     expr, expression to put in place of varName
122         * @post      Expression may change, references and iterators
```

```
123          *            remain valid after this function.
124          **/
125          void mapVariable( string varName, ExpressionAtom expr );
126
127          /**
128          * @function  mapVariables( )
129          * @abstract  Same as mapVariable( ) for a set of variables/expressions
130          * @param     varmap, list of varName/expr pairs
131          * @post      Expression may change, references and iterators
132          *            remain valid after this function.
133          **/
134          void mapVariables( const map<string,ExpressionAtom>& varmap );
135
136          /**
137          * @function  generateInOrder( )
138          * @abstract  generates the infix notation of the tree.
139          * @param     out, the way in which we want to see the output
140          * @post      The infix notation of the tree has been generated
141          **/
142          void generateInOrder( ostream& out ) const {
143              generateInOrderRecursive( m_root, out );
144          }
145
146      private:
147          /**
148          * @function  differentiate( ), differentiateExponent( ),
149          *            differentiateDivision( ), differentiateProduct( ),
150          *            differentiateFunction( ), differentiateAddition( )
151          * @abstract  differentiates ExpressionTree and places the derivative in
152          *            the tree assigned to the last variable
153          * @param     n, the node we need to start differentiating from
154          * @param     varName, variable name to match (e.g, 'x')
155          * @param     derivative, the node we want to differentiate from
156          * @param     derivatree, the tree we want to differentiate to
157          * @return    the derivative of the original function in the form of a
158          *            tree
159          * @pre       There needs to be a tree
160          * @post      The derivatree has been changed, now it shows the
161          *            derivative of ExpressionTree.
162          **/
163          void differentiate( node_t * n, string varName,
164                              node_t * derivative,
165                              ExpressionTree &derivatree );
166          void differentiateExponent( node_t * n, string varName,
167                                      node_t * derivative,
168                                      ExpressionTree &derivatree );
169          void differentiateDivision( node_t * n, string varName,
170                                      node_t * derivative,
171                                      ExpressionTree &derivatree );
172          void differentiateProduct( node_t * n, string varName,
173                                     node_t * derivative,
174                                     ExpressionTree &derivatree );
175          void differentiateFunction( node_t * n, string varName,
176                                      node_t * derivative,
```

```
177                                     ExpressionTree &derivatree );
178         void differentiateAddition( node_t * n, string varName,
179                                     node_t * derivative,
180                                     ExpressionTree &derivatree );
181
182         /**
183          * @function  simplify( )
184          * @abstract  Performs mathematical simplification on the expression
185          * @param     root, root of the subtree to simplify
186          * @return    New node in place of the passed value/node for root
187          * @post      Upon simplification, nodes may be deleted.
188          *            references and iterators may become invalid
189          **/
190         node_t *simplifyRecursive( node_t* root );
191
192         /**
193          * @function  generateInOrderRecursive( )
194          * @abstract  Recursively goes through the tree to get the infix notation
195          *            of the tree
196          * @param     root, the node we're looking at
197          * @param     buffer, the output
198          * @post      Eventually the infix notation of the tree with parenthesis
199          *            has been generated.
200          **/
201         void generateInOrderRecursive( node_t *root, ostream& buffer ) const;
202
203     public:
204         /**
205          * @function  compare( )
206          * @abstract  Throws a parser expression.
207          * @param     f1, the first value we want to compare
208          * @param     f2, the second value we want to compare
209          * @param     error, the marge in which the difference is accepted.
210          * @return    if the difference between f1 and f2 is smaller or equal to
211          *            error
212          * @post      A ParserException is thrown.
213          **/
214         static bool compare( const float &f1, const float &f2, float &&error =0.00001
215             return ( fabs( f1-f2 ) <= error );
216         }
217
218 };
219 #endif
```

ExpressionTree.cc

```
1  /**
2   * ExpressionTree:
3   *
4   * @author  Micky Faas (s1407937)
5   * @author  Lisette de Schipper (s1396250)
6   * @file    ExpressionTree.cc
7   * @date    26-10-2014
8   **/
```

```
9
10 #include "ExpressionTree.h"
11
12 list<ExpressionAtom> ExpressionTree::tokenize( const string& str ) {
13
14     list<ExpressionAtom> tokenlist;
15     stringstream ss( str );
16     while( ss.good( ) ) {
17         string token;
18         ss >> token;
19         ExpressionAtom atom;
20         bool unary_minus =false;
21
22         if( token.size( ) > 1 && token[0] == '-' ) {
23             token =token.substr( 1 );
24             unary_minus =true;
25         }
26
27         if( token.find( "." ) != string::npos ) { // Float
28             try {
29                 atom.setFloat( (unary_minus ? -1.0f : 1.0f)
30                                * std::stof( token ) );
31                 unary_minus =false;
32             } catch( std::invalid_argument& e ) {
33                 throw ParserException( string ("Invalid float '")
34                                        + token
35                                        + string("'") );
36             }
37         }
38         else if( token == "*" )
39             atom.setOperator( ExpressionAtom::PRODUCT );
40         else if( token == "/" )
41             atom.setOperator( ExpressionAtom::DIVISION );
42         else if( token == "+" )
43             atom.setOperator( ExpressionAtom::SUM );
44         else if( token == "-" )
45             atom.setOperator( ExpressionAtom::DIFFERENCE );
46         else if( token == "^" )
47             atom.setOperator( ExpressionAtom::EXPONENT );
48         else if( token == "sin" )
49             atom.setFunction( ExpressionAtom::SIN );
50         else if( token == "cos" )
51             atom.setFunction( ExpressionAtom::COS );
52         else if( token == "tan" )
53             atom.setFunction( ExpressionAtom::TAN );
54         else if( token == "ln" )
55             atom.setFunction( ExpressionAtom::LN );
56         else if( token == "log" )
57             atom.setFunction( ExpressionAtom::LOG );
58         else if( token == "sqrt" )
59             atom.setFunction( ExpressionAtom::SQRT );
60         else if( token == "abs" )
61             atom.setFunction( ExpressionAtom::ABS );
62         else if( token == "e" )
```

```
63              atom.setFunction( ExpressionAtom::E );
64          else if( token == "pi" )
65              atom.setFunction( ExpressionAtom::PI );
66          else if( token.find( "/" ) != string::npos ) { // Fraction
67              size_t pos =token.find( "/" );
68              Fraction f;
69              try {
70                  f.numerator =(unary_minus ? -1 : 1)
71                              * std::stoi( token.substr( 0, pos ) );
72                  f.denominator =std::stoi( token.substr( pos + 1 ) );
73                  atom.setFraction( f );
74                  unary_minus =false;
75              }
76              catch( std::invalid_argument& e ){
77                  throw ParserException( string ("Invalid fraction `")
78                                         + token
79                                         + string("`") );
80              }
81          }
82          else {
83              try { // Try integer
84                  atom.setInteger( (unary_minus ? -1 : 1) * std::stol( token ) );
85                  unary_minus =false;
86
87              } // Try variable
88              catch( invalid_argument& e ){
89                  for( unsigned int i =0; i < token.size( ); ++i )
90                      if( !isalpha( token[i] ) )
91                          throw ParserException( string ("Invalid token `")
92                                                 + token
93                                                 + string("`") );
94                  atom.setNamed( token );
95              }
96          }
97
98          if( unary_minus )
99              tokenlist.push_back( ExpressionAtom::UNARY_MINUS );
100         tokenlist.push_back( atom );
101     }
102     return tokenlist;
103 }
104
105 void ExpressionTree::fromString( const string& expression ) {
106     list<ExpressionAtom> tokenlist;
107
108     try{
109         tokenlist =ExpressionTree::tokenize( expression );
110     } catch( ParserException & e ) {
111         throw e;
112     }
113
114     Tree<ExpressionAtom>::node_t *n =0;
115
116     for( auto atom : tokenlist ) {
```

```
117             if( !n ) {
118                 n =pushBack( atom );
119                 continue;
120             }
121             while ( !n->info( ).arity( )
122             || ( n->info( ).arity( ) == 1 && n->hasChildren( ) )
123             || ( n->info( ).arity( ) == 2 && n->isFull( ) ) ) {
124                 n =n->parent ( );
125                 if( !n )
126                     throw ParserException( "Argument count to arity mismatch" );
127             }
128 
129             n =insert( atom, n );
130         }
131     }
132 
133     ExpressionTree ExpressionTree::differentiate( string varName ) {
134         ExpressionTree derivatree;
135         differentiate( root( ), varName, derivatree.root( ), derivatree );
136         derivatree.simplify( );
137         return derivatree;
138     }
139 
140     void ExpressionTree::simplify( ) {
141         m_root =simplifyRecursive( root( ) );
142     }
143 
144     ExpressionTree
145     ExpressionTree::evaluate( string varName, ExpressionAtom expr ) const {
146         ExpressionTree t( *this );
147         t.mapVariable( varName, expr );
148         t.simplify( );
149         return std::move( t );
150     }
151 
152     ExpressionTree
153     ExpressionTree::evaluate( const map<string,ExpressionAtom>& varmap ) const {
154         ExpressionTree t( *this );
155         t.mapVariables( varmap );
156         t.simplify( );
157         return std::move( t );
158     }
159 
160     void ExpressionTree::mapVariable( string varName, ExpressionAtom expr ) {
161         map<string,ExpressionAtom> varmap;
162         varmap[varName] =expr;
163         mapVariables( varmap );
164     }
165 
166     void ExpressionTree::mapVariables( const map<string,ExpressionAtom>& varmap ) {
167         for( auto &node : *this ) {
168             if( node.info( ).type( ) == ExpressionAtom::NAMED_OPERAND ) {
169                 auto it =varmap.find( node.info( ).getNamed( ) );
170                 if( it != varmap.cend( ) )
```

```
171                 node =it->second;
172         }
173     }
174 }
175
176 void ExpressionTree::differentiate( node_t * n, string varName,
177                                     node_t * derivative,
178                                     ExpressionTree &derivatree ) {
179     ExpressionAtom atom =(*n);
180     switch( atom.type( ) ) {
181         case ExpressionAtom::OPERATOR:
182         switch( atom.getOperator( ) ) {
183             case ExpressionAtom::SUM:
184             case ExpressionAtom::DIFFERENCE:
185                 differentiateAddition( &(*n), varName, derivative, derivatree );
186                 break;
187             case ExpressionAtom::PRODUCT:
188                 differentiateProduct( &(*n), varName, derivative, derivatree );
189                 break;
190             case ExpressionAtom::EXPONENT:
191                 differentiateExponent( &(*n), varName, derivative, derivatree );
192                 break;
193             case ExpressionAtom::DIVISION:
194                 differentiateDivision( &(*n), varName, derivative, derivatree );
195                 break;
196         }
197         break;
198         case ExpressionAtom::FUNCTION:
199             differentiateFunction( &(*n), varName, derivative, derivatree );
200             break;
201         case ExpressionAtom::NAMED_OPERAND:
202             atom.getNamed( ) == string( varName ) ?
203             derivatree.insert( 1L, derivative ) :
204             derivatree.insert( 0L, derivative );
205             break;
206         default:
207             derivatree.insert( 0L, derivative );
208     }
209 }
210
211 void ExpressionTree::differentiateFunction( node_t * n, string varName,
212                                             node_t * derivative,
213                                             ExpressionTree &derivatree ) {
214     Tree<ExpressionAtom> tempTree;
215     Tree<ExpressionAtom>::node_t *temp;
216     ExpressionAtom atom =(*n);
217     switch( atom.getFunction( ) ){
218         case ExpressionAtom::SIN:
219             temp =derivatree.insert( ExpressionAtom::PRODUCT, derivative );
220             differentiate( (*n).leftChild( ), varName, temp, derivatree );
221             temp =derivatree.insert( ExpressionAtom::COS, temp );
222             copyFromNode( (*n).leftChild( ), temp, true );
223             break;
224         case ExpressionAtom::TAN:;
```

```
225             temp =tempTree.insert( ExpressionAtom::DIVISION, tempTree.root( ) );
226             temp =tempTree.insert( ExpressionAtom::SIN, temp );
227             copyFromNode( (*n).leftChild( ), temp, true );
228             temp =temp->parent( );
229             temp =tempTree.insert( ExpressionAtom::COS, temp );
230             copyFromNode( (*n).leftChild( ), temp, true );
231             differentiate( tempTree.root( ), varName, derivative, derivatree );
232             tempTree.clear( );
233             break;
234         case ExpressionAtom::COS:
235             temp =derivatree.insert( ExpressionAtom::PRODUCT, derivative );
236             temp =derivatree.insert( ExpressionAtom::UNARY_MINUS, temp );
237             differentiate( (*n).leftChild( ), varName, temp, derivatree );
238             temp =temp->parent( );
239             temp =derivatree.insert( ExpressionAtom::SIN, temp );
240             copyFromNode( (*n).leftChild( ), temp, true );
241             break;
242         case ExpressionAtom::LN:
243             if( contains( (*n).leftChild( ), string( varName ) ) ) {
244                 temp =derivatree.insert( ExpressionAtom::DIVISION, derivative);
245                 differentiate( (*n).leftChild( ), varName, temp, derivatree );
246                 copyFromNode( (*n).leftChild( ), temp, false );
247             }
248             else
249                 derivatree.insert( 0L, derivative);
250             break;
251         case ExpressionAtom::SQRT:
252             temp =derivatree.insert( ExpressionAtom::DIVISION, derivative );
253             differentiate( (*n).leftChild( ), varName, temp, derivatree );
254             temp =derivatree.insert( ExpressionAtom::PRODUCT, temp );
255             derivatree.insert( 2L, temp );
256             copyFromNode( &(*n), temp, false );
257             break;
258         case ExpressionAtom::LOG:
259             temp =derivatree.insert( ExpressionAtom::PRODUCT, derivative );
260             temp =derivatree.insert( ExpressionAtom::DIVISION, temp );
261             derivatree.insert( 1L, temp);
262             temp =derivatree.insert( ExpressionAtom::LN, temp );
263             copyFromNode( (*n).leftChild( ), temp, true );
264             temp =temp->parent( )->parent( );
265             temp =derivatree.insert( ExpressionAtom::DIVISION, temp );
266             differentiate( (*n).rightChild( ), varName, temp, derivatree );
267             copyFromNode( (*n).rightChild( ), temp, false );
268             break;
269         case ExpressionAtom::ABS:
270             if( (*n).leftChild( )->info( ).type( ) ==
271                 ExpressionAtom::NAMED_OPERAND &&
272                 (*n).leftChild( )->info( ).getNamed( ) == string( varName ) ) {
273                 temp =derivatree.insert( ExpressionAtom::DIVISION, derivative );
274                 copyFromNode( (*n).leftChild( ), temp, true );
275                 copyFromNode( &(*n), temp, false );
276             }
277             else {
278                 temp =derivatree.insert( ExpressionAtom::DIVISION, derivative );
```

```
279                 temp =derivatree.insert( ExpressionAtom::PRODUCT, temp );
280                 copyFromNode( (*n).leftChild( ), temp, true );
281                 differentiate( (*n).leftChild( ), varName, temp, derivatree );
282                 temp =temp->parent( );
283                 copyFromNode( &(*n), temp, false );
284             }
285             break;
286     }
287 }
288
289 void ExpressionTree::differentiateAddition( node_t * n, string varName,
290                                             node_t * derivative,
291                                             ExpressionTree &derivatree ) {
292     Tree<ExpressionAtom>::node_t *temp;
293     ExpressionAtom atom =(*n);
294     if( atom.getOperator( ) == ExpressionAtom::SUM )
295         temp =derivatree.insert( ExpressionAtom::SUM, derivative );
296     else
297         temp =derivatree.insert( ExpressionAtom::DIFFERENCE, derivative );
298     differentiate( (*n).leftChild( ), varName, temp, derivatree );
299     if( (*n).rightChild( ) )
300         differentiate( (*n).rightChild( ), varName, temp, derivatree );
301 }
302
303 void ExpressionTree::differentiateDivision( node_t * n, string varName,
304                                             node_t * derivative,
305                                             ExpressionTree &derivatree ) {
306     Tree<ExpressionAtom>::node_t *temp;
307     temp =derivatree.insert( ExpressionAtom::DIVISION, derivative );
308     temp =derivatree.insert( ExpressionAtom::DIFFERENCE, temp );
309     temp =derivatree.insert( ExpressionAtom::PRODUCT, temp);
310     copyFromNode( (*n).rightChild( ), temp, true );
311     differentiate( (*n).leftChild( ), varName, temp, derivatree );
312     temp =temp->parent( );
313     temp =derivatree.insert( ExpressionAtom::PRODUCT, temp);
314     copyFromNode( (*n).leftChild( ), temp, true );
315     differentiate( (*n).rightChild( ), varName, temp, derivatree );
316     temp =temp->parent( )->parent( );
317     temp =derivatree.insert( ExpressionAtom::EXPONENT, temp );
318     copyFromNode( (*n).rightChild( ), temp, true );
319     derivatree.insert( 2L, temp );
320 }
321
322 void ExpressionTree::differentiateProduct( node_t * n, string varName,
323                                            node_t * derivative,
324                                            ExpressionTree &derivatree ) {
325     Tree<ExpressionAtom>::node_t *temp;
326     if( (*n).leftChild( )->info( ).isNumericOperand( ) ) {
327         // n * x
328         if( (*n).rightChild( )->info( ).type( ) ==
329             ExpressionAtom::NAMED_OPERAND &&
330             (*n).rightChild( )->info( ).getNamed( ) == string( varName ) )
331             derivatree.insert( (*n).leftChild( )->info( ), derivative );
332         // n * f(x)
```

```
333             else {
334                 temp =derivatree.insert( ExpressionAtom::PRODUCT, derivative );
335                  derivatree.insert( (*n).leftChild( )->info( ), temp );
336                  differentiate( (*n).rightChild( ), varName, temp, derivatree );
337             }
338         }
339         else if( (*n).rightChild( )->info( ).isNumericOperand( ) ) {
340             // x * n
341             if( (*n).leftChild( )->info( ).type( ) ==
342                 ExpressionAtom::NAMED_OPERAND &&
343                 (*n).leftChild( )->info( ).getNamed( ) == string( varName ) )
344                 derivatree.insert( (*n).rightChild( )->info( ), derivative );
345             // f(x) * n
346             else {
347                 temp =derivatree.insert( ExpressionAtom::PRODUCT, derivative );
348                  derivatree.insert( (*n).rightChild( )->info( ), temp );
349                  differentiate( (*n).leftChild( ), varName, temp, derivatree );
350             }
351         }
352         // f(x) * g(x)
353         else {
354             temp =derivatree.insert( ExpressionAtom::SUM, derivative );
355             temp =derivatree.insert( ExpressionAtom::PRODUCT, temp );
356             copyFromNode( (*n).rightChild( ), temp, true );
357             differentiate( (*n).leftChild( ), varName, temp, derivatree );
358             temp =temp->parent( );
359             temp =derivatree.insert( ExpressionAtom::PRODUCT, temp );
360             copyFromNode( (*n).leftChild( ), temp, true );
361             differentiate( (*n).rightChild( ), varName, temp, derivatree );
362         }
363     }
364
365     void ExpressionTree::differentiateExponent( node_t * n, string varName,
366                                                 node_t * derivative,
367                                                 ExpressionTree &derivatree ) {
368         Tree<ExpressionAtom>::node_t *temp;
369         Tree<ExpressionAtom> tempTree;
370         if( contains( (*n).leftChild( ), string( varName ) ) ) {
371             // f(x) ^ g(x)
372             if( contains( (*n).rightChild( ), string( varName ) ) ) {
373                 // f(x)^g(x) =e^(ln(f(x))g(x))
374                 temp =tempTree.insert( ExpressionAtom::EXPONENT, tempTree.root( ) );
375                 tempTree.insert( ExpressionAtom::E, temp );
376                 temp =tempTree.insert( ExpressionAtom::PRODUCT, temp );
377                 temp =tempTree.insert( ExpressionAtom::LN, temp );
378                 copyFromNode( (*n).leftChild( ), temp, true );
379                 temp =temp->parent( );
380                 copyFromNode( (*n).rightChild( ), temp, false );
381                 differentiate( tempTree.root( ), varName, derivative, derivatree );
382                 tempTree.clear( );
383             }
384             // f(x) ^ n
385             else {
386                 if( (*n).leftChild( )->info( ).type( ) ==
```

```
387                 ExpressionAtom::NAMED_OPERAND &&
388                 (*n).leftChild( )->info( ).getNamed( ) ==
389                 string( varName )  ) {
390                 // x ^ 0
391                 if( (*n).rightChild( )->info( )  == 0L )
392                     derivatree.insert( 1L, derivative );
393                 // x ^ 1
394                 else if( (*n).rightChild( )->info( ) == 1L )
395                     derivatree.insert( string( "x" ), derivative );
396                 // x ^ n ( n > 1 )
397                 else if( (*n).rightChild( )->info( ) > 1L ) {
398                     temp =derivatree.insert( ExpressionAtom::PRODUCT,
399                                              derivative );
400                     derivatree.insert( (*n).rightChild( )->info( ), temp );
401                     temp =derivatree.insert( ExpressionAtom::EXPONENT , temp );
402                     derivatree.insert( string( varName ) , temp );
403                     derivatree.insert( (*n).rightChild( )->info( ) - 1L, temp );
404                 }
405                 // x ^ n ( n < 0 )
406                 else if( (*n).rightChild( )->info( ) < 0L ) {
407                     temp =derivatree.insert( ExpressionAtom::DIVISION,
408                                              derivative);
409                     derivatree.insert( (*n).rightChild( )->info( ), temp );
410                     temp =derivatree.insert( ExpressionAtom::EXPONENT, temp);
411                     derivatree.insert( string( varName ), temp);
412                     derivatree.insert( (*n).rightChild( )->info( ) -
413                                        (*n).rightChild( )->info( ) -
414                                        (*n).rightChild( )->info( ) + 1L, temp);
415                 }
416             }
417             else {
418                 temp =derivatree.insert( ExpressionAtom::PRODUCT, derivative );
419                 temp =derivatree.insert( ExpressionAtom::PRODUCT, temp );
420                 copyFromNode( (*n).rightChild( ), temp, true );
421                 temp =derivatree.insert( ExpressionAtom::EXPONENT, temp );
422                 copyFromNode( (*n).leftChild( ), temp, true );
423                 derivatree.insert( (*n).rightChild( )->info( ) -
424                                    (*n).rightChild( )->info( ) - 1L, temp );
425                 temp =temp->parent( )->parent( );
426                 differentiate( (*n).leftChild( ), varName, temp, derivatree );
427             }
428         }
429     }
430     //e ^ f(x)
431     else if( (*n).leftChild( )->info( ).type( ) == ExpressionAtom::FUNCTION &&
432              (*n).leftChild( )->info( ).getFunction( ) == ExpressionAtom::E ) {
433         temp =derivatree.insert( ExpressionAtom::PRODUCT, derivative) ;
434         differentiate( (*n).rightChild( ), varName, temp, derivatree );
435         copyFromNode( &(*n), temp, false);
436     }
437     // n ^ f(x)
438     else if( contains( (*n).rightChild( ), string( varName ) ) ) {
439         temp =derivatree.insert( ExpressionAtom::PRODUCT, derivative );
440         temp =derivatree.insert( ExpressionAtom::PRODUCT, temp );
```

```
441         differentiate( (*n).rightChild( ), varName, temp, derivatree );
442         temp =derivatree.insert( ExpressionAtom::LN, temp );
443         copyFromNode( (*n).leftChild( ), temp, true );
444         temp =temp->parent( )->parent( );
445         temp =derivatree.insert( ExpressionAtom::EXPONENT, temp );
446         copyFromNode( (*n).leftChild( ), temp, true );
447         copyFromNode( (*n).rightChild( ), temp, false );
448     }
449 }
450
451
452 ExpressionTree::node_t *
453 ExpressionTree::simplifyRecursive( node_t* root ) {
454     if( !root )
455         return 0;
456
457     node_t *n =root->leftChild( );
458     node_t *m =root->rightChild( );
459
460     /* cascade( ): removes root and child n, replaces root with child m */
461     auto cascade =[&]( ) -> node_t* {
462         remove( n );
463         if( root->parent( ) ) {
464             if( root ==root->parent( )->leftChild( ) )
465                 root->parent( )->setLeftChild( m );
466             else
467                 root->parent( )->setRightChild( m );
468             m->setParent( root->parent( ) );
469         }
470         else
471             m->setParent( 0 );
472         delete root;
473         return m;
474     };
475
476     /* merge( ):
477        replaces the root by the result of its operation on the children */
478     auto merge =[&]( ) -> node_t* {
479
480         ExpressionAtom &lhs =root->leftChild( )->info( );
481         ExpressionAtom &rhs =root->rightChild( )->info( );
482         ExpressionAtom &op =root->info( );
483
484         assert( lhs.isNumericOperand( ) && rhs.isNumericOperand( ) );
485
486         switch( op.getOperator( ) ) {
487             case ExpressionAtom::SUM:
488                 op =std::move( lhs + rhs );
489                 break;
490             case ExpressionAtom::DIFFERENCE:
491                 op =std::move( lhs - rhs );
492                 break;
493             case ExpressionAtom::PRODUCT:
494                 op =std::move( lhs * rhs );
```

```
495                 break;
496             case ExpressionAtom::DIVISION:
497                 op =std::move( lhs / rhs );
498                 break;
499             case ExpressionAtom::EXPONENT:
500                 op =std::move( lhs.pow( rhs ) );
501                 break;
502         }
503 
504         remove( m );
505         remove( n );
506         return root;
507     };
508 
509     /* mergeInto( ): replaces the root by expr and removes the children */
510     auto mergeInto =[&]( ExpressionAtom&& expr ) -> node_t* {
511         remove( m );
512         remove( n );
513         root->info( ) =std::move( expr );
514         return root;
515     };
516 
517     bool stop =false;
518     do {
519 
520         if( n ) {
521             n =simplifyRecursive( n );
522             if( n && !n->hasChildren( ) ) {
523                 // Simplify the one-fraction
524                 if( n->info( ).type( ) == ExpressionAtom::FRACTION_OPERAND
525                         && n->info( ).getFraction( ).numerator == 1 )
526                     n->info( ).setInteger( 1 );
527 
528                 // two operands-case
529                 if( n->info( ).isNumericOperand( )
530                     && m && m->info( ).isNumericOperand( ) ) {
531                     root =merge( );
532                     return root;
533                 }
534 
535                 // 1 case
536                 if( n->info( ).isNumericOperand( )
537                     && compare( 1.0f, n->info( ).toFloat( ) ) ) {
538                     if( root->info( ) == ExpressionAtom::PRODUCT ) {
539                         root =cascade( );
540                     }
541                     else if( root->info( ) == ExpressionAtom::EXPONENT ) {
542                         if( n == root->leftChild( ) )
543                             root =mergeInto( 1l );
544                         else
545                             root =cascade( );
546                     }
547                     else if( root->info( ) == ExpressionAtom::DIVISION ) {
548                         if( n == root->rightChild( ) )
```

```
549                             root =cascade( );
550                         }
551                     }
552                     // 0 case
553                     else if( n->info( ).isNumericOperand( )
554                             && compare( 0.0f, n->info( ).toFloat( ) ) ) {
555                         if( root->info( ) == ExpressionAtom::SUM )
556                             root =cascade( );
557                         else if( root->info( ) == ExpressionAtom::PRODUCT ) {
558                             root =mergeInto( 0l );
559                         }
560                         else if( root->info( ) == ExpressionAtom::DIVISION ) {
561                             if( n == root->leftChild( ) )
562                                 root =mergeInto( 0l );
563                         }
564                         else if( root->info( ) == ExpressionAtom::DIFFERENCE ) {
565                             if( n == root->rightChild( ) )
566                                 root =cascade( );
567                             else if( m && m->info( ).isNumericOperand( ) ) {
568                                 root =mergeInto( ExpressionAtom( -1l )
569                                                  * m->info( ) );
570                             }
571                         }
572                         else if( root->info( ) == ExpressionAtom::EXPONENT ) {
573                             if( n == root->leftChild( ) ) {
574                                 if( m && m->info( ).isNumericOperand( )
575                                       && compare( 1.0f, m->info( ).toFloat( ) ) )
576                                     root =mergeInto( 1l );
577                                 else {
578                                     root =mergeInto( 0l );
579                                 }
580                             }
581                             else {
582                                 root =mergeInto( 1l );
583                             }
584                         }
585                     }
586                     // trivial functions
587                     else if( root->info( ).type( ) == ExpressionAtom::FUNCTION ) {
588                         switch( root->info( ).getFunction( ) ) {
589                             case ExpressionAtom::UNARY_MINUS:
590                                 if( n->info( ).isNumericOperand( ) )
591                                     root =mergeInto( ExpressionAtom( -1l )
592                                                      * n->info( ) );
593                                 break;
594                             case ExpressionAtom::LN: // ln(e)
595                                 if( n->info( ) == ExpressionAtom::E )
596                                     root =mergeInto( 1l );
597                                 break;
598                         }
599                     }
600                 }
601             }
602
```

```
603         if( stop )
604             break;
605 
606         n =root->rightChild( );
607         m =root->leftChild( );
608         stop =true;
609     } while( n );
610 
611 return root;
612 }
613 
614 void
615 ExpressionTree::generateInOrderRecursive( node_t *root, ostream& buffer ) const{
616     if( !root )
617         return;
618 
619     if( root->info( ).type( ) == ExpressionAtom::FUNCTION ) {
    // Function type
620         bool enclose =root->isFull( ) // Only enclose in ( )'s if neccessary
621             || ( root->leftChild( ) && !root->leftChild( )->hasChildren( ) )
622             || ( root->rightChild( ) && !root->rightChild( )->hasChildren( ) );
623 
624         if( root->info( ).getFunction( ) == ExpressionAtom::UNARY_MINUS ) {
625             buffer << '(';
626             enclose =false;
627         }
628 
629         buffer << root->info( );
630 
631         if( enclose )
632             buffer << '(';
633 
634         generateInOrderRecursive( root->leftChild( ), buffer );
635 
636         if( root->isFull( ) ) // Function with two params, otherwise no comma
637             buffer << ',';
638 
639         generateInOrderRecursive( root->rightChild( ), buffer );
640 
641         if( enclose )
642             buffer << ')';
643 
644         if( root->info( ).getFunction( ) == ExpressionAtom::UNARY_MINUS )
645             buffer << ')';
646     } else {    // Operator+operands type
647         if( root->hasChildren( ) && root != m_root )
648             buffer << '(';
649 
650         generateInOrderRecursive( root->leftChild( ), buffer );
651 
652         if( !(root->info( ) == ExpressionAtom::PRODUCT  // implicit multipl.
653             && root->leftChild( )
654             && root->leftChild( )->info( ).isNumericOperand( ) ) )
655             buffer << root->info( );
```

```
656         generateInOrderRecursive( root->rightChild( ), buffer );
657
658         if( root->hasChildren( ) && root != m_root )
659             buffer << ')';
660     }
661 }
```

main.cc

```
1  /**
2   * main.cc: Simpel programma dat de functionaliteit
3   *   van ExpressionTree demonstreert
4   *
5   * @author  Micky Faas (s1407937)
6   * @author  Lisette de Schipper (s1396250)
7   * @file    main.cc
8   * @date    26-10-2014
9   **/
10
11 #include <iostream>
12 #include "ExpressionTree.h"
13 #include <string>
14
15 using namespace std;
16
17 /**
18 * @function  showEvaluation( )
19 * @abstract  subinterface for evaluation of a tree
20 * @param     tree, the tree we want to evaluate
21 * @post      the tree is evaluated
22 **/
23 void showEvaluation( ExpressionTree& tree ) {
24     string var;
25     float value;
26
27     cout << "What is the variable?" << endl;
28     getline( cin, var );
29     cout << "What is the value you want to fill in?" << endl;
30     cin >> value;
31     cout << "It has been evaluated to: " << endl;
32
33     tree.evaluate( var, value ).generateInOrder( cout );
34
35     cout << endl;
36 }
37
38 /**
39 * @function  saveToDot( )
40 * @abstract  subinterface for the conversion to Dot-notation of a tree
41 * @param     tree, the tree we want to convert
42 * @post      the tree is converted
43 **/
44 void saveToDot( ExpressionTree& tree ) {
45     string input;
```

```
46      ofstream file;
47
48      cout << "To what file should the tree be written?" << endl;
49      getline( cin, input );
50      file.open( input );
51      cout << "How should we call the tree?" << endl;
52      getline( cin, input );
53
54      tree.toDot( file, input );
55      cout << "Done!" << endl;
56
57  }
58
59  int main ( ) {
60      string input;
61      string moreInput;
62      char inputChar;
63      char inputTree;
64      ExpressionTree expression;
65      ExpressionTree derivative;
66
67      cout << "With this program you can differentiate an expression. "
68           << "The program has been made by Lisette and Micky. Enjoy!" << endl
69           << endl
70           << "What's the expression in prefix notation?" << endl;
71      getline( cin, input );
72      expression =input;
73
74      while( inputChar != 'q') {
75          cout << "Do you want to [d]ifferentiate, [s]implify, [e]valuate, "
76               << "[c]onvert to Dot or [q]uit?" << endl;
77          cin >> inputChar;
78          cin.ignore( );
79          switch( inputChar ) {
80              case 'c':
81              case 'e':
82                  if( !derivative.isEmpty( ) ) {
83                      cout << "The [d]erivative or the [o]riginal tree?" << endl;
84                      cin >> inputTree;
85                      cin.ignore( );
86                      if( ! (inputChar == 'o' ||  inputChar == 'd') )
87                          cout << "Invalid Input." << endl;
88
89                  }
90                  else
91                      inputTree ='o';
92
93                  if( inputChar == 'c' )
94                      saveToDot( inputTree == 'o' ? expression : derivative );
95                  else
96                      showEvaluation( inputTree == 'o' ? expression : derivative );
97                  break;
98
99              case 'd':
```

```
100                 cout << "What is the variable?" << endl;
101                 getline( cin, input );
102                 derivative =expression.differentiate( input );
103                 cout << "The tree has been derived to :";
104                 derivative.generateInOrder( cout );
105                 cout << endl;
106                 break;
107             case 's':
108                 expression.simplify( );
109                 cout << "the tree has been simplified to : ";
110                 expression.generateInOrder( cout );
111                 cout << endl;
112                 break;
113             case 'q':
114                 cout << "Thank you for having used this program. Goodbye."
115                     << endl;
116                 break;
117             default:
118                 cout << "You have entered an invalid character." << endl;
119         }
120     }
121     return 0;
122 }
```

Tree.h

```
1  /**
2   * Tree:
3   *
4   * @author  Micky Faas (s1407937)
5   * @author  Lisette de Schipper (s1396250)
6   * @file    tree.h
7   * @date    26-10-2014
8   **/
9
10 #ifndef TREE_H
11 #define TREE_H
12 #include "TreeNodeIterator.h"
13 #include <assert.h>
14 #include <list>
15 #include <map>
16
17 using namespace std;
18
19 template <class INFO_T> class Tree
20 {
21     public:
22         enum ReplaceBehavoir {
23             DELETE_EXISTING,
24             ABORT_ON_EXISTING,
25             MOVE_EXISTING
26         };
27
28         typedef TreeNode<INFO_T> node_t;
```

```
29          typedef TreeNodeIterator<INFO_T> iterator;
30          typedef TreeNodeIterator_in<INFO_T> iterator_in;
31          typedef TreeNodeIterator_pre<INFO_T> iterator_pre;
32          typedef TreeNodeIterator_post<INFO_T> iterator_post;
33          typedef list<node_t*> nodelist;
34
35         /**
36          * @function  Tree( )
37          * @abstract  Constructor of a tree
38          **/
39          Tree( )
40              : m_root( 0 ) {
41          }
42
43         /**
44          * @function  Tree( )
45          * @abstract  Constructor of a tree. The tree becomes the tree given as
46          *            the parameter
47          * @param     tree, a tree
48          **/
49          Tree( const Tree<INFO_T>& tree )
50              : m_root( 0 ) {
51              *this =tree;
52          }
53
54          /**
55          * @function  ~Tree( )
56          * @abstract  Destructor of a tree. Timber.
57          **/
58          ~Tree( ) {
59            clear( );
60          }
61
62         /**
63          * @function  begin_pre( )
64          * @abstract  begin point for pre-order iteration
65          * @return    interator_pre containing the beginning of the tree in
66          *            pre-order
67          **/
68          iterator_pre begin_pre( ) {
69              // Pre-order traversal starts at the root
70              return iterator_pre( m_root );
71            }
72
73         /**
74          * @function  begin( )
75          * @abstract  begin point for a pre-order iteration
76          * @return    containing the beginning of the pre-0rder iteration
77          **/
78          iterator_pre begin( ) {
79              return begin_pre( );
80          }
81
82         /**
```

```
83          * @function  end( )
84          * @abstract  end point for a pre-order iteration
85          * @return    the end of the pre-order iteration
86          **/
87          iterator_pre end( ) {
88              return iterator_pre( (node_t*)0 );
89          }
90
91         /**
92          * @function  end_pre( )
93          * @abstract  end point for pre-order iteration
94          * @return    interator_pre containing the end of the tree in pre-order
95          **/
96          iterator_pre end_pre( ) {
97              return iterator_pre( (node_t*)0 );
98          }
99
100        /**
101         * @function  begin_in( )
102         * @abstract  begin point for in-order iteration
103         * @return    interator_in containing the beginning of the tree in
104         *            in-order
105         **/
106         iterator_in begin_in( ) {
107             if( !m_root )
108                 return end_in( );
109             node_t *n =m_root;
110             while( n->leftChild( ) )
111                 n =n->leftChild( );
112             return iterator_in( n );
113           }
114
115        /**
116         * @function  end_in( )
117         * @abstract  end point for in-order iteration
118         * @return    interator_in containing the end of the tree in in-order
119         **/
120         iterator_in end_in( ) {
121             return iterator_in( (node_t*)0 );
122         }
123
124        /**
125         * @function  begin_post( )
126         * @abstract  begin point for post-order iteration
127         * @return    interator_post containing the beginning of the tree in
128         *            post-order
129         **/
130         iterator_post begin_post( ) {
131             if( !m_root )
132                 return end_post( );
133             node_t *n =m_root;
134             while( n->leftChild( ) )
135                 n =n->leftChild( );
136             return iterator_post( n );
```

```
137         }
138
139        /**
140         * @function  end_post( )
141         * @abstract  end point for post-order iteration
142         * @return    interator_post containing the end of the tree in post-order
143         **/
144         iterator_post end_post( ) {
145             return iterator_post( (node_t*)0 );
146         }
147
148        /**
149         * @function  pushBack( )
150         * @abstract  a new TreeNode containing 'info' is added to the end
151         *            the node is added to the node that :
152         *               - is in the row as close to the root as possible
153         *               - has no children or only a left-child
154         *               - seen from the right hand side of the row
155         *            this is the 'natural' left-to-right filling order
156         *            compatible with array-based heaps and full b-trees
157         * @param     info, the contents of the new node
158         * @post      A node has been added.
159         **/
160         node_t *pushBack( const INFO_T& info ) {
161             node_t *n =new node_t( info, 0 );
162             if( !m_root ) { // Empty tree, simplest case
163                 m_root =n;
164             }
165             else { // Leaf node, there are two different scenarios
166                 int max =getRowCountRecursive( m_root, 0 );
167                 node_t *parent;
168                 for( int i =1; i <= max; ++i ) {
169
170                     parent =getFirstEmptySlot( i );
171                     if( parent ) {
172                         if( !parent->leftChild( ) )
173                             parent->setLeftChild( n );
174                         else if( !parent->rightChild( ) )
175                             parent->setRightChild( n );
176                         n->setParent( parent );
177                         break;
178                     }
179                 }
180             }
181             return n;
182         }
183
184        /**
185         * @function  insert( )
186         * @abstract  inserts node or subtree under a parent or creates an empty
187         *            root node
188         * @param     info, contents of the new node
189         * @param     parent, parent node of the new node. When zero, the root is
190         *            assumed
```

```
191         * @param     alignRight, insert() checks on which side of the parent
192         *            node the new node can be inserted. By default, it checks
193         *            the left side first.
194         *            To change this behavior, set preferRight =true.
195         * @param     replaceBehavior, action if parent already has two children.
196         *            One of:
197         *            ABORT_ON_EXISTING - abort and return zero
198         *            MOVE_EXISTING - make the parent's child a child of the new
199         *                            node, satisfies preferRight
200         *            DELETE_EXISTING - remove one of the children of parent
201         *                              completely also satisfies preferRight
202         * @return    pointer to the inserted TreeNode, if insertion was
203         *            successfull
204         * @pre       If the tree is empty, a root node will be created with info
205         *            as it contents
206         * @pre       The instance pointed to by parent should be part of the
207         *            called instance of Tree
208         * @post      Return zero if no node was created. Ownership is assumed on
209         *            the new node.
210         *            When DELETE_EXISTING is specified, the entire subtree on
211         *            preferred side may be deleted first.
212         **/
213        node_t* insert( const INFO_T& info,
214                        node_t* parent =0,
215                        bool preferRight =false,
216                        int replaceBehavior =ABORT_ON_EXISTING ) {
217            if( !parent )
218                parent =m_root;
219
220            if( !parent )
221                return pushBack( info );
222
223            node_t *node =0;
224
225            if( !parent->leftChild( )
226                  && ( !preferRight || ( preferRight &&
227                        parent->rightChild( ) ) ) ) {
228                node =new node_t( info, parent );
229                parent->setLeftChild( node );
230                node->setParent( parent );
231
232            } else if( !parent->rightChild( ) ) {
233                node =new node_t( info, parent );
234                parent->setRightChild( node );
235                node->setParent( parent );
236
237            } else if( replaceBehavior == MOVE_EXISTING ) {
238                node =new node_t( info, parent );
239                if( preferRight ) {
240                    node->setRightChild( parent->rightChild( ) );
241                    node->rightChild( )->setParent( node );
242                    parent->setRightChild( node );
243                } else {
244                    node->setLeftChild( parent->leftChild( ) );
```

```
245                     node->leftChild( )->setParent( node );
246                     parent->setLeftChild( node );
247                 }
248 
249             } else if( replaceBehavior == DELETE_EXISTING ) {
250                 node =new node_t( info, parent );
251                 if( preferRight ) {
252                     deleteRecursive( parent->rightChild( ) );
253                     parent->setRightChild( node );
254                 } else {
255                     deleteRecursive( parent->leftChild( ) );
256                     parent->setLeftChild( node );
257                 }
258 
259             }
260             return node;
261         }
262 
263         /**
264          * @function  replace( )
265          * @abstract  replaces an existing node with a new node
266          * @param     info, contents of the new node
267          * @param     node, node to be replaced. When zero, the root is assumed
268          * @param     alignRight, only for MOVE_EXISTING. If true, node will be
269          *            the right child of the new node. Otherwise, it will be the
270          *            left.
271          * @param     replaceBehavior, one of:
272          *            ABORT_ON_EXISTING - undefined for replace()
273          *            MOVE_EXISTING - make node a child of the new node,
274          *                            satisfies preferRight
275          *            DELETE_EXISTING - remove node completely
276          * @return    pointer to the inserted TreeNode, replace() is always
277          *            successful
278          * @pre       If the tree is empty, a root node will be created with info
279          *            as it contents
280          * @pre       The instance pointed to by node should be part of the
281          *            called instance of Tree
282          * @post      Ownership is assumed on the new node. When DELETE_EXISTING
283          *            is specified, the entire subtree pointed to by node is
284          *            deleted first.
285          **/
286         node_t* replace( const INFO_T& info,
287                          node_t* node =0,
288                          bool alignRight =false,
289                          int replaceBehavior =DELETE_EXISTING ) {
290             assert( replaceBehavior != ABORT_ON_EXISTING );
291 
292             node_t *newnode =new node_t( info );
293             if( !node )
294                 node =m_root;
295             if( !node )
296                 return pushBack( info );
297 
298             if( node->parent( ) ) {
```

```
299                 newnode->setParent( node->parent( ) );
300                 if( node->parent( )->leftChild( ) == node )
301                     node->parent( )->setLeftChild( newnode );
302                 else
303                     node->parent( )->setRightChild( newnode );
304             } else
305                 m_root =newnode;
306 
307             if( replaceBehavior == DELETE_EXISTING ) {
308 
309                 deleteRecursive( node );
310             }
311             else if( replaceBehavior == MOVE_EXISTING ) {
312                 if( alignRight )
313                     newnode->setRightChild( node );
314                 else
315                     newnode->setLeftChild( node );
316                 node->setParent( newnode );
317             }
318             return node;
319         }
320 
321         /**
322          * @function  remove( )
323          * @abstract  removes and deleltes node or subtree
324          * @param     n, node or subtree to be removed and deleted
325          * @post      after remove(), n points to an invalid address
326          **/
327          void remove( node_t *n ) {
328             if( !n )
329                 return;
330             if( n->parent( ) ) {
331                 if( n->parent( )->leftChild( ) == n )
332                     n->parent( )->setLeftChild( 0 );
333                 else if( n->parent( )->rightChild( ) == n )
334                     n->parent( )->setRightChild( 0 );
335             }
336             deleteRecursive( n );
337         }
338 
339         /**
340          * @function  clear( )
341          * @abstract  clears entire tree
342          * @pre       tree may be empty
343          * @post      all nodes and data are deallocated
344          **/
345          void clear( ) {
346             deleteRecursive( m_root );
347             m_root =0;
348         }
349 
350         /**
351          * @function  empty( )
352          * @abstract  test if tree is empty
```

```
353          * @return    true when empty
354          **/
355          bool isEmpty( ) const {
356              return !m_root;
357          }
358
359        /**
360          * @function  root( )
361          * @abstract  returns address of the root of the tree
362          * @return    the adress of the root of the tree is returned
363          * @pre       there needs to be a tree
364          **/
365          node_t* root( ){
366              return m_root;
367          }
368
369         /**
370          * @function  row( )
371          * @abstract  returns an entire row/level in the tree
372          * @param     level, the desired row. Zero gives just the root.
373          * @return    a list containing all node pointers in that row
374          * @pre       level must be positive or zero
375          * @post
376          **/
377          nodelist row( int level ) {
378              nodelist rlist;
379              getRowRecursive( m_root, rlist, level );
380              return rlist;
381          }
382
383         /**
384          * @function  contains( )
385          * @abstract  find the first occurrence of info and returns its node ptr
386          * @param     haystack, the root of the (sub)tree we want to look in
387          * @param     needle, the needle in our haystack
388          * @return    a pointer to the first occurrence of needle
389          * @post      there may be multiple occurrences of needle, we only return
390          *            one. A null-pointer is returned if no needle is found
391          **/
392          node_t* contains( node_t* haystack, const INFO_T& needle ) {
393              if( haystack == 0 ) {
394                      if( m_root )
395                          haystack =m_root;
396                      else
397                          return 0;
398              }
399              return findRecursive( haystack, needle );
400          }
401
402         /**
403          * @function  toDot( )
404          * @abstract  writes tree in Dot-format to a stream
405          * @param     out, ostream to write to
406          * @pre       out must be a valid stream
```

```
407          * @post      out (file or cout) with the tree in dot-notation
408          **/
409          void toDot( ostream& out, const string & graphName ) {
410              if( isEmpty( ) )
411                  return;
412              map< node_t *, int> adresses;
413              typename map< node_t *, int >::iterator adrIt;
414              int i =1;
415              int p;
416              iterator_pre it;
417              iterator_pre tempit;
418              adresses[m_root] =0;
419              out << "digraph " << graphName << '{' << endl << '"' << 0 << '"';
420              for( it =begin_pre( ); it != end_pre( ); ++it ) {
421                  adrIt =adresses.find( &(*it) );
422                  if( adrIt == adresses.end( ) ) {
423                      adresses[&(*it)] =i;
424                      p =i;
425                      i ++;
426                  }
427                  if( (&(*it))->parent( ) != &(*tempit) )
428                    out << ';' << endl << '"'
429                        << adresses.find( (&(*it))->parent( ))->second << '"';
430                  if( (&(*it)) != m_root )
431                      out << " ->  \"" << p << '"';
432                  tempit =it;
433              }
434              out << ';' << endl;
435              for ( adrIt =adresses.begin( ); adrIt != adresses.end( ); ++adrIt )
436                  out << adrIt->second << " [label=\""
437                      << adrIt->first->info( ) << "\"]";
438              out << '}';
439          }
440
441         /**
442          * @function  copyFromNode( )
443          * @abstract  copies the the node source and its children to the node
444          *            dest
445          * @param     source, the node and its children that need to be copied
446          * @param     dest, the node who is going to get the copied children
447          * @param     left, this is true if it's a left child.
448          * @pre       there needs to be a tree and we can't copy to a root.
449          * @post      the subtree that starts at source is now also a child of
450          *            dest
451          **/
452          void copyFromNode( node_t *source, node_t *dest, bool left ) {
453              node_t *acorn =new node_t( dest );
454              if(left) {
455                  if( dest->leftChild( ))
456                      return;
457                  dest->setLeftChild( acorn );
458              }
459              else {
460                  if( dest->rightChild( ))
```

```
461                     return;
462                 dest->setRightChild( acorn );
463             }
464             cloneRecursive( source, acorn );
465         }
466 
467         Tree<INFO_T>& operator=( const Tree<INFO_T>& tree ) {
468             clear( );
469             if( tree.m_root ) {
470                 m_root =new node_t( (node_t*)0 );
471                 cloneRecursive( tree.m_root, m_root );
472             }
473             return *this;
474         }
475 
476 
477     private:
478        /**
479         * @function  cloneRecursive( )
480         * @abstract  cloning a subtree to a node
481         * @param     source, the node we want to start the cloning process from
482         * @param     dest, the node we want to clone to
483         * @post      the subtree starting at source is cloned to the node dest
484         **/
485         void cloneRecursive( node_t *source, node_t* dest ) {
486             dest->info() =source->info();
487             if( source->leftChild( ) ) {
488                 node_t *left =new node_t( dest );
489                 dest->setLeftChild( left );
490                 cloneRecursive( source->leftChild( ), left );
491             }
492             if( source->rightChild( ) ) {
493                 node_t *right =new node_t( dest );
494                 dest->setRightChild( right );
495                 cloneRecursive( source->rightChild( ), right );
496             }
497         }
498 
499        /**
500         * @function  deleteRecursive( )
501         * @abstract  delete all nodes of a given tree
502         * @param     root, starting point, is deleted last
503         * @post      the subtree has been deleted
504         **/
505         void deleteRecursive( node_t *root ) {
506             if( !root )
507                 return;
508             deleteRecursive( root->leftChild( ) );
509             deleteRecursive( root->rightChild( ) );
510             delete root;
511         }
512 
513        /**
514         * @function  getRowCountRecursive( )
```

```
515          * @abstract  calculate the maximum depth/row count in a subtree
516          * @param     root, starting point
517          * @param     level, starting level
518          * @return    maximum depth/rows in the subtree
519          **/
520          int getRowCountRecursive( node_t* root, int level ) {
521              if( !root )
522                  return level;
523              return max(
524                      getRowCountRecursive( root->leftChild( ), level+1 ),
525                      getRowCountRecursive( root->rightChild( ), level+1 ) );
526          }
527
528         /**
529          * @function  getRowRecursive( )
530          * @abstract  compile a full list of one row in the tree
531          * @param     root, starting point
532          * @param     rlist, reference to the list so far
533          * @param     level, how many level still to go
534          * @post      a list of a row in the tree has been made.
535          **/
536          void getRowRecursive( node_t* root, nodelist &rlist, int level ) {
537              // Base-case
538              if( !level ) {
539                  rlist.push_back( root );
540              } else if( root ){
541                  level--;
542                  if( level && !root->leftChild( ) )
543                      for( int i =0; i < (level<<1); ++i )
544                          rlist.push_back( 0 );
545                  else
546                     getRowRecursive( root->leftChild( ), rlist, level );
547
548                  if( level && !root->rightChild( ) )
549                      for( int i =0; i < (level<<1); ++i )
550                          rlist.push_back( 0 );
551                  else
552                      getRowRecursive( root->rightChild( ), rlist, level );
553              }
554          }
555
556          /**
557          * @function  getFirstEmptySlot( )
558          * @abstract  when a row has a continuous empty space on the right,
559          *            find the left-most parent in the above row that has
560          *            at least one empty slot.
561          * @param     level, how many level still to go
562          * @return    the first empty slot where we can put a new node
563          * @pre       level should be > 1
564          **/
565          node_t *getFirstEmptySlot( int level ) {
566              node_t *p =0;
567              nodelist rlist =row( level-1 ); // we need the parents of this level
568              /** changed auto to int **/
```

```
569             for( auto it =rlist.rbegin( ); it !=rlist.rend( ); ++it ) {
570                 if( !(*it)->hasChildren( ) )
571                     p =(*it);
572                 else if( !(*it)->rightChild( ) ) {
573                     p =(*it);
574                     break;
575                 } else
576                     break;
577             }
578             return p;
579         }
580 
581         /**
582         * @function  findRecursive( )
583         * @abstract  first the first occurrence of needle and return its node
584         *            ptr
585         * @param     haystack, root of the search tree
586         * @param     needle, copy of the data to find
587         * @return    the node that contains the needle
588         **/
589         node_t *findRecursive( node_t* haystack, const INFO_T &needle ) {
590             if( haystack->info( ) == needle )
591                 return haystack;
592 
593             node_t *n =0;
594             if( haystack->leftChild( ) )
595                 n =findRecursive( haystack->leftChild( ), needle );
596             if( !n && haystack->rightChild( ) )
597                 n =findRecursive( haystack->rightChild( ), needle );
598             return n;
599         }
600 
601         friend class TreeNodeIterator_pre<INFO_T>;
602         friend class TreeNodeIterator_in<INFO_T>;
603     protected:
604         TreeNode<INFO_T> *m_root;
605 };
606 
607 #endif
```

TreeNode.h

```
1  /**
2   * Treenode:
3   *
4   * @author  Micky Faas (s1407937)
5   * @author  Lisette de Schipper (s1396250)
6   * @file    Treenode.h
7   * @date    26-10-2014
8   **/
9 
10 #ifndef TREEINFO_T_H
11 #define TREEINFO_T_H
12 
```

```
13  using namespace std;
14
15  template <class INFO_T> class Tree;
16  class ExpressionTree;
17
18  template <class INFO_T> class TreeNode
19  {
20      public:
21         /**
22          * @function  TreeNode( )
23          * @abstract  Constructor, creates a node
24          * @param     info, the contents of a node
25          * @param     parent, the parent of the node
26          * @post      A node has been created.
27          **/
28          TreeNode( const INFO_T& info, TreeNode<INFO_T>* parent =0 )
29              : m_lchild( 0 ), m_rchild( 0 ) {
30              m_info =info;
31              m_parent =parent;
32          }
33
34         /**
35          * @function  TreeNode( )
36          * @abstract  Constructor, creates a node
37          * @param     parent, the parent of the node
38          * @post      A node has been created.
39          **/
40          TreeNode( TreeNode<INFO_T>* parent =0 )
41              : m_lchild( 0 ), m_rchild( 0 ) {
42              m_parent =parent;
43          }
44
45         /**
46          * @function  =
47          * @abstract  Sets a nodes content to N
48          * @param     n, the contents you want the node to have
49          * @post      The node now has those contents.
50          **/
51          void operator =( INFO_T n ) { m_info =n; }
52
53         /**
54          * @function  INFO_T( ), info( )
55          * @abstract  Returns the content of a node
56          * @return    m_info, the contents of the node
57          **/
58          operator INFO_T( ) const { return m_info; }
59          const INFO_T &info( ) const { return m_info; }
60          INFO_T &info( ) { return m_info; }
61          /**
62          * @function  atRow( )
63          * @abstract  returns the level or row-number of this node
64          * @return    row, an int of row the node is at
65          **/
66          int atRow( ) const {
```

```
67              const TreeNode<INFO_T> *n =this;
68              int row =0;
69              while( n->parent( ) ) {
70                  n =n->parent( );
71                  row++;
72              }
73              return row;
74          }
75
76          /**
77           * @function  parent( ), leftChild( ), rightChild( )
78           * @abstract  returns the adress of the parent, left child and right
79           *            child respectively
80           * @return    the adress of the requested family member of the node
81           **/
82          TreeNode<INFO_T> *parent( ) const { return m_parent; }
83          TreeNode<INFO_T> *leftChild( ) const { return m_lchild; }
84          TreeNode<INFO_T> *rightChild( ) const { return m_rchild; }
85
86          /**
87           * @function  sibling( )
88           * @abstract  returns the address of the sibling
89           * @return    the address to the sibling or zero if there is no sibling
90           **/
91          TreeNode<INFO_T>* sibling( ) {
92              if( parent( )->leftChild( ) == this )
93                  return parent( )->rightChild( );
94              else if( parent( )->rightChild( ) == this )
95                  return parent( )->leftChild( );
96              else
97                  return 0;
98          }
99
100         /**
101          * @function  hasChildren( ), hasParent( ), isFull( )
102          * @abstract  Returns whether the node has children, has parents or is
103          *            full (has two children) respectively
104          * @param
105          * @return    true or false, depending on what is requested from the node.
106          *            if hasChildren is called and the node has children, it will
107          *            return true, otherwise false.
108          *            If hasParent is called and the node has a parent, it will
109          *            return true, otherwise false.
110          *            If isFull is called and the node has two children, it will
111          *            return true, otherwise false.
112          **/
113         bool hasChildren( ) const { return m_lchild || m_rchild; }
114         bool hasParent( ) const { return m_parent; }
115         bool isFull( ) const { return m_lchild && m_rchild; }
116
117     protected:
118         friend class Tree<INFO_T>;
119         friend class ExpressionTree;
120
```

```
121         /**
122          * @function  setParent( ), setLeftChild( ), setRightChild( )
123          * @abstract  sets the parent, left child and right child of the
124          *            particular node respectively
125          * @param     p, the node we want to set a certain family member of
126          * @return    void
127          * @post      The node now has a parent, a left child or a right child
128          *            respectively.
129          **/
130          void setParent( TreeNode<INFO_T> *p ) { m_parent =p; }
131          void setLeftChild( TreeNode<INFO_T> *p ) { m_lchild =p; }
132          void setRightChild( TreeNode<INFO_T> *p ) { m_rchild =p; }
133
134      private:
135          INFO_T m_info;
136          TreeNode<INFO_T> *m_parent;
137          TreeNode<INFO_T> *m_lchild;
138          TreeNode<INFO_T> *m_rchild;
139  };
140
141  /**
142  * @function  <<
143  * @abstract  the contents of the node are returned
144  * @param     out, in what format we want to get the contents
145  * @param     rhs, the node of which we want the contents
146  * @return    the contents of the node.
147  **/
148  template <class INFO_T> ostream &operator <<(ostream& out, const TreeNode<INFO_T> & r
149      out << rhs.info( );
150      return out;
151  }
152
153  #endif
```

TreeNodeIterator.h

```
1   /**
2    * TreeNodeIterator: Provides a set of iterators that follow the STL-standard
3    *
4    * @author  Micky Faas (s1407937)
5    * @author  Lisette de Schipper (s1396250)
6    * @file    TreeNodeIterator.h
7    * @date    26-10-2014
8    **/
9
10  #include <iterator>
11  #include "TreeNode.h"
12
13  template <class INFO_T> class TreeNodeIterator
14                          : public std::iterator<std::forward_iterator_tag,
15                                                 TreeNode<INFO_T>> {
16      public:
17          typedef TreeNode<INFO_T> node_t;
18
```

```
19          /**
20           * @function  TreeNodeIterator( )
21           * @abstract  (copy)constructor
22           * @pre       TreeNodeIterator is abstract and cannot be constructed
23           **/
24           TreeNodeIterator( node_t* ptr =0 ) : p( ptr ) { }
25           TreeNodeIterator( const TreeNodeIterator& it ) : p( it.p ) { }
26
27          /**
28           * @function  (in)equality operator overload
29           * @abstract  Test (in)equality for two TreeNodeIterators
30           * @param     rhs, right-hand side of the comparison
31           * @return    true if both iterators point to the same node (==)
32           *            false if both iterators point to the same node (!=)
33           **/
34           bool operator == (const TreeNodeIterator& rhs) { return p==rhs.p; }
35           bool operator != (const TreeNodeIterator& rhs) { return p!=rhs.p; }
36
37          /**
38           * @function  operator*( )
39           * @abstract  Cast operator to node_t reference
40           * @return    The value of the current node
41           * @pre       Must point to a valid node
42           **/
43           node_t& operator*( ) { return *p; }
44
45          /**
46           * @function  operator++( )
47           * @abstract  pre- and post increment operators
48           * @return    TreeNodeIterator that has iterated one step
49           **/
50           TreeNodeIterator &operator++( ) { next( ); return *this; }
51           TreeNodeIterator operator++( int )
52               { TreeNodeIterator tmp( *this ); operator++( ); return tmp; }
53      protected:
54
55          /**
56           * @function  next( ) (pure virtual)
57           * @abstract  Implement this function to implement your own iterator
58           */
59           virtual bool next( ) =0;
60           node_t *p;
61  };
62
63  template <class INFO_T> class TreeNodeIterator_pre
64                          : public TreeNodeIterator<INFO_T> {
65      public:
66           typedef TreeNode<INFO_T> node_t;
67
68           TreeNodeIterator_pre( node_t* ptr =0 )
69               : TreeNodeIterator<INFO_T>( ptr ) { }
70           TreeNodeIterator_pre( const TreeNodeIterator<INFO_T>& it )
71               : TreeNodeIterator<INFO_T>( it ) { }
72           TreeNodeIterator_pre( const TreeNodeIterator_pre& it )
```

```
73              : TreeNodeIterator<INFO_T>( it.p ) { }
74
75          TreeNodeIterator_pre &operator++( ) { next( ); return *this; }
76          TreeNodeIterator_pre operator++( int )
77              { TreeNodeIterator_pre tmp( *this ); operator++( ); return tmp; }
78
79      protected:
80          using TreeNodeIterator<INFO_T>::p;
81
82         /**
83          * @function  next( )
84          * @abstract  Takes one step in pre-order traversal
85          * @return    returns true if such a step exists
86          */
87          bool next( ) {
88              if( !p )
89                  return false;
90              if( p->hasChildren( ) ) { // a possible child that can be the next
91                  p =p->leftChild( ) ? p->leftChild( ) : p->rightChild( );
92                  return true;
93              }
94              else if( p->hasParent( ) // we have a right brother
95                      && p->parent( )->rightChild( )
96                      && p->parent( )->rightChild( ) != p ) {
97                  p =p->parent( )->rightChild( );
98                  return true;
99              }
100             else if( p->hasParent( ) ) { // just a parent, thus we go up
101                 TreeNode<INFO_T> *tmp =p->parent( );
102                 while( tmp->parent( ) ) {
103                     if( tmp->parent( )->rightChild( )
104                             && tmp->parent( )->rightChild( ) != tmp ) {
105                         p =tmp->parent( )->rightChild( );
106                         return true;
107                     }
108                     tmp =tmp->parent( );
109                 }
110             }
111             // Nothing left
112             p =0;
113             return false;
114         }
115
116 };
117
118 template <class INFO_T> class TreeNodeIterator_in
119                         : public TreeNodeIterator<INFO_T>{
120     public:
121         typedef TreeNode<INFO_T> node_t;
122
123         TreeNodeIterator_in( node_t* ptr =0 )
124             : TreeNodeIterator<INFO_T>( ptr ) { }
125         TreeNodeIterator_in( const TreeNodeIterator<INFO_T>& it )
126             : TreeNodeIterator<INFO_T>( it ) { }
```

```
127         TreeNodeIterator_in( const TreeNodeIterator_in& it )
128             : TreeNodeIterator<INFO_T>( it.p ) { }
129 
130         TreeNodeIterator_in &operator++( ) { next( ); return *this; }
131         TreeNodeIterator_in operator++( int )
132             { TreeNodeIterator_in tmp( *this ); operator++( ); return tmp; }
133 
134     protected:
135         using TreeNodeIterator<INFO_T>::p;
136        /**
137         * @function  next( )
138         * @abstract  Takes one step in in-order traversal
139         * @return    returns true if such a step exists
140         */
141         bool next( ) {
142             if( p->rightChild( ) ) {
143                 p =p->rightChild( );
144                 while( p->leftChild( ) )
145                     p =p->leftChild( );
146                 return true;
147             }
148             else if( p->parent( ) && p->parent( )->leftChild( ) == p ) {
149                 p =p->parent( );
150                 return true;
151             } else if( p->parent( ) && p->parent( )->rightChild( ) == p ) {
152                 p =p->parent( );
153                 while( p->parent( ) && p == p->parent( )->rightChild( ) ) {
154                     p =p->parent( );
155                 }
156                 if( p )
157                     p =p->parent( );
158                 if( p )
159                     return true;
160                 else
161                     return false;
162             }
163             // Er is niks meer
164             p =0;
165             return false;
166         }
167 };
168 
169 template <class INFO_T> class TreeNodeIterator_post
170                         : public TreeNodeIterator<INFO_T>{
171     public:
172         typedef TreeNode<INFO_T> node_t;
173 
174         TreeNodeIterator_post( node_t* ptr =0 )
175             : TreeNodeIterator<INFO_T>( ptr ) { }
176         TreeNodeIterator_post( const TreeNodeIterator<INFO_T>& it )
177             : TreeNodeIterator<INFO_T>( it ) { }
178         TreeNodeIterator_post( const TreeNodeIterator_post& it )
179             : TreeNodeIterator<INFO_T>( it.p ) { }
180 
```

```
181         TreeNodeIterator_post &operator++( ) { next( ); return *this; }
182         TreeNodeIterator_post operator++( int )
183             { TreeNodeIterator_post tmp( *this ); operator++( ); return tmp; }
184 
185     protected:
186         using TreeNodeIterator<INFO_T>::p;
187        /**
188         * @function  next( )
189         * @abstract  Takes one step in post-order traversal
190         * @return    returns true if such a step exists
191         */
192         bool next( ) {
193 
194             if( p->hasParent( ) // We have a right brother
195                     && p->parent( )->rightChild( )
196                     && p->parent( )->rightChild( ) != p ) {
197                 p =p->parent( )->rightChild( );
198                 while( p->leftChild( ) )
199                     p =p->leftChild( );
200                 return true;
201             } else if( p->parent( ) ) {
202                 p =p->parent( );
203                 return true;
204             }
205             // Nothing left
206             p =0;
207             return false;
208         }
209 };
```