```
1  /*=============================================================================
2      Copyright (c) 2002-2003 Joel de Guzman
3      http://spirit.sourceforge.net/
4
5      Use, modification and distribution is subject to the Boost Software
6      License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
7      http://www.boost.org/LICENSE_1_0.txt)
8  =============================================================================*/
9
10 #include "includes.hpp"
11
12
13
14 #include <boost/spirit.hpp>
15 #include <iostream>
16 #include <string>
17
18
19
20 using namespace std;
21 // Plus pratique pour appeller les fonctions de l'API spirit.
22 using namespace boost::spirit;
23
24
25
26 // Actions sémantiques exécutées sur trigger (reconnaissance d'un objet par exemple)
27 // Ce ne sont que des exemples.
28 namespace
29 {
30     void    do_int(char const* str, char const* end)
31     {
32         string  s(str, end);
33         cout << "INT(" << s << ')' << endl;
34
35     }
36
37
38     void    do_var(char const* str, char const* end)
39     {
40         string  s(str, end);
41         cout << "VAR(" << s << ')' << endl;
42     }
43
44
45     void    do_add(char const*, char const*)
46     {
47         cout << "ADD\n";
48     }
49
50
51     void    do_subt(char const*, char const*)
52     {
53         cout << "SUBTRACT\n";
54     }
55
56
57     void    do_mult(char const*, char const*)
58     {
59         cout << "MULTIPLY\n";
```

```
60      }
61
62
63      void    do_div(char const*, char const*)
64      {
65          cout << "DIVIDE\n";
66      }
67
68
69      void    do_neg(char const*, char const*)
70      {
71          cout << "NEGATE\n";
72      }
73
74
75      // Un arbre s'adaptant à n'importe quel node
76      template<typename CNode> class MyTree
77      {
78          public:
79              CNode node;
80
81              // Lien vers les fils
82              vector<CNode*> children;
83      };
84
85      MyTree<int> root;
86
87      // Vars est un vecteur de type Variable
88      vector<Variable> args;
89
90  }
91
92  ////////////////////////////////////////////////////////////////////////////
    /
93  //
94  //  Our calculator grammar
95  //
96  ////////////////////////////////////////////////////////////////////////////
    /
97
98  // On se définit une classe publique calculator qui dérive de grammar
99  struct calculator : public grammar<calculator>
100 {
101     // Elle contient une structure interne nommee definition (convention)
102     // Il s'agit d'une template sur un type ScannerT
103     template <typename ScannerT> struct definition
104     {
105
106         // Constructeur qui se refere à son conteneur
107         definition(calculator const& self)
108         {
109
110             // Un entier : lexeme_d passe en mode parsing caractère. Plante
     sur des espaces.
111             integer =
112                 lexeme_d[ (+digit_p)[&do_int] ]
113                 ;
114
115             // une declaration de vars est un lexeme_d d'un caractere puis
    une suite de cars alphanums ou _ .
116             // La declaration est retenue uniquement si la variable n'exist
    e pas.
```

```
117             var_decl =
118                 lexeme_d
119                 [
120                     ( +alpha_p >> *( alnum_p | '_' ) )[&do_var]
121                 ]
122                 ;
123 
124             /*args =
125                 while_p( integer | var_decl )[&args.push_back*/
126 
127             // Une fonction
128             //function =
129 
130 
131 
132 
133             // Un facteur : un entier ou une variable ou une expression ent
    re parentheses, ou +/- un autre facteur
134             factor =
135                 integer |
136                 //function
137                 var_decl |
138                 '(' >> expression >> ')' |
139                 ( '-' >> factor )[&do_neg] |
140                 ( '+' >> factor )
141                 ;
142 
143             // un TERME est un facteur que multiplie ou divise d'autres fac
    teurs
144             term =
145                 factor
146                 >> *( ( '*' >> factor)[&do_mult]
147                     | ( '/' >> factor)[&do_div]
148                 )
149                 ;
150 
151             // Une expression est une addition d'au moins 2 termes.
152             expression =
153                 term >> *(
154                         ( '+' >> term)[&do_add]
155                       | ( '-' >> term)[&do_subt]
156                  )
157                 ;
158 
159         }
160 
161 
162 
163         // Chaque element de la grammaire est une regle parametree par notr
    e scanner.
164         rule<ScannerT> expression, term, factor, integer, var_decl;
165 
166         // Symbole de demarrage de la grammaire
167         rule<ScannerT> const& start() const { return expression; }
168     };
169 };
170 
171 
172 
```