NAME

Properties, proptest

SYNOPSIS

~nmoor0/CS216/bin/proptest [*filename*]

AVAILABILITY

This program may be found on cslab, in the directory ~nmoor0/CS216/bin/.

ASSOCIATED FILES

Properties.man (this file) may be found in ~nmoor0/CS216/man/ on any cslab machine. A test file, proptest.prop, may be found in ~nmoor0/CS216/misc/, again on any cslab machine.

DESCRIPTION

proptest treats its argument as a filename; if not provided with an argument, it defaults to "proptest.prop". Each line of that file should follow one of the following formats:

[*ws*]*property*[*ws*]=[*ws*][*value*]
[*ws*] #*comment*
[*ws*]

Each *property* begins with an alphabetic character, followed by any number of alphanumeric characters and/or dots ("."). Two or more dots may not appear consecutively, and the property name may not end with a dot.

The property list is loaded from the file into memory, where it may be operated upon by the user. Each property has associated with it a string value (which may be empty). **proptest** presents the user with a menu of operations. These include printing the entire property list to the screen, saving it to a file, retrieving the value of a particular property, testing for the existence of a property, adding (or updating) a property, and removing a property. When the user enters a command, **proptest** prompts the user for the needed parameters, tests for the validity of these parameters, and performs the operation requested. The program handles errors somewhat gracefully; it displays a concise error message and returns to the prompt.

RETURN VALUE

0 if things went normally, 1 otherwise. Currently, there are no fatal errors, so proptest will always return 0.

NOTES ON IMPLEMENTATION

proptest is intended as a driver program for the **Properties** class; this class implements a properties manager. **proptest** itself is rather simple. The **main()** function consists of a loop which reads a command from stdin, dispatches that command to a **docommand()** function, and loops (assuming the user did not type "quit"). **docommand()** reads additional parameters from stdin and calls the appropriate function in the **Properties** class. It catches any exceptions that may be thrown (so that the program does not abruptly terminate).

Properties consists of a number of methods. These include constructors, a destructor, the assignment operator, **load**(), a number of property retrieval functions, **addProperty**(), **remove**(), **isDefined**(), **save(),**, and **valid_property**(). In addition, there is the associated **ostream <<** operator.

load(), as the name suggests, loads into memory a property file (in the format outlined in the **DESCRIPTION** section above), the filename for which is indicated by its **const char *** argument. Properties in memory are stored as a linked list of (*property*, *value*) pairs (this implementation detail may be changed in the future; STL **map**s are a quite attractive alternative). Each line is parsed by **match_prop()** (a protected static member function—see below). **load()** prints a warning message to stderr when it encounters an invalid line or property name. This behaviour would ideally be generalised, such that **Properties**'s constructor accepted as an argument a function object, which would be called for each error (we cannot use exceptions, as we wish to continue processing further lines in the event of an error). Such a behaviour may be implemented in a future version of this class.

The **Properties** class contains a number of functions to retrieve the value of a property. The most basic of these is **getValueFor()**, which takes as its argument a property name. It returns the value associated with that property (always a string); if the property is undefined, it throws a **NotFound** exception (see below). There are other functions (**getBoolValueFor()**, **getIntValueFor()**, and **getDoubleValueFor()**) which interpret the value of a property as a particular type (**bool**, **int**, and **double**, respectively). If the property does not fit the type requested, these methods throw a **BadFormat** exception (see below). **getIntValueFor()** interprets values as **strtol()** would (a leading “0x” indicates hexadecimal, a leading “0” octal). **getDoubleValueFor()** uses **strtod()** to make the conversion (thus accepting scientific notation such as “1.3E-4”). Finally, **getBoolValueFor()** interprets “1”, “on”, or “true” (any capitalisation) as **true**, and “0”, “off”, or “false” as **false**.

The **addProperty()** method adds a (**property**, **value**) pair to the property list. It uses **valid_property()** (see below) to test the validity of the property name. If the property name is invalid, it throws a **BadProperty** exception. If the indicated property already exists in the property list, **addProperty()** updates its value—the same property may not appear twice in the list.

remove(), surprisingly enough, removes a pair (specified by property name) from the property list. If the specified property does not exist, **remove()** throws a **NotFound** exception.

isDefined() is quite simple; it returns **true** if the specified property is in the property list, **false** otherwise. Because the property list is stored as a standard linked list, this method (as well as most of the others) is O(n) with respect to the number of properties in the list.

save() writes the property list to a file suitable for use with **Properties::load()**. It utilises **ostream operator<<()** (see below) to do the actual output. Note that **load()**ing a file the **save()**ing it loses information (such as comments, blank lines, and redefinitions).

valid_property() (a static member function) simply tests the validity of a property name. The proper syntax for property names is described in the **DESCRIPTION** section above. This method basically tacks '=' to the end of the property name, and hands the result off to **match_prop()**. **match_prop()** itself is a protected static member function. It attempts to match the property part of a *property*=*value* line. This function matches the ed-style regular expression:

```
^[ \t]*\([a-z][a-z0-9]*(\.[a-z0-9]+)*\)[ \t]*=
```

This regexp corresponds to optional leading whitespace, followed by at least one alphabetic character, followed by any number of alphanumeric characters, followed by optional whitespace and a necessary equals sign; dots may be interspersed within the alphanumeric sequence, but two dots may not occur adjacently, and the last character of that sequence must not be a dot.
The text enclosed within `\(...\)` (which corresponds to the property name) is placed in the string to which **match_prop()**'s second parameter refers. The value returned by **match_prop()** indicates the total number of characters matched (including whitespace and the equals sign). If the line is invalid, it returns 0 and sets its second (string reference) argument to the empty string.

Rather than using a more obvious ad-hoc parser, **match_prop()** uses a deterministic finite automaton

(DFA) to match the string. We do not implement a full regular expression handler, however: the state table for the above regexp is hard-coded into the program. The algorithm here is basically:

1 *state* = 0; *prop* = the empty string; (*prop* is the reference argument which will hold the matched property name.)

2 if we are at the end of the string, return failure

3 *in* = code representing the type of the next character (alphabetic, numeric, dot, whitespace, equals sign, or other).

4 **state** = *next_state*(*state*, *in*)

5 if *state* is a failing state, return failure

6 if *state* is an accepting state, return success

7 if *in* is not 'whitespace', append the current character to *prop*.

8 Go to step 2

Note that this algorithm differs from **grep**(**1**) and **ed**(**1**) with respect to certain parts of its matching. Notably, it quits as soon as it reaches an accepting state, rather than trying to find a maximal match.

readPropLine() is a protected static member function which reads a line from an **istream** and breaks it into property and value components. It is a wrapper around **match_prop()**. **writePropLine** is also a protected static member function. It writes a specified (*property*, *value*) pair from an **ostream** in the format expected by **readPropLine()**. **ostream &operator<<()** is a friend function of the **Properties** class. It basically calls **writePropLine()** for each pair in the property list.

There are also a few miscellaneous functions defined in and/or used by **Properties**. These include: **chartypeof()** (a private static member function, called only by match_prop, which converts a **char** into a character code of the type mentioned in the **match_prop()** algorithm above; **stripws()** (in util.cc), which removes leading and trailing whitespace from a **std::string**; and **lowercasify()**, which, oddly enough, converts a **std::string** to lowercase.

Various member functions of the **Properties** class may throw exceptions. These exceptions are classes defined within the scope of **Properties** (so they are referred to as **Properties::***ExceptionName*). The exception classes are:

- **CannotOpen**, thrown by **load()**, **save()**, and one of the constructors. This exception indicates that a file could not be opened in the appropriate mode. **CannotOpen::errno()** gives the value of **errno** when the failure was detected; **CannotOpen::filename()** gives the filename that could not be opened.

- **BadProperty**, thrown by **addProperty()**. This exception indicates that the name of a property is invalid (according to **valid_property**()). **BadProperty::propname()** gives the invalid property in question.

- **BadFormat**, thrown by **getBoolValueFor()**, **getIntValueFor()**, and **getDoubleValueFor()**. This exception indicates that, although the requested property exists, its value does not match the requested type. For example, calling **getIntValueFor**("foo") when the value of foo is "purple" will result in a **BadFormat** exception being thrown. **BadFormat::propname()** gives the name of the property in question; **BadFormat::value()** gives the (string) value of that property.

- **NotFound**, thrown by **get*ValueFor()** and **remove()**. This exception indicates that an undefined property was referred to. **NotFound::propname()** gives the name of the property which was referred to.

There is also an internal-only **BadLine** exception, thrown by **readPropLine** (which, as pointed out above, is a protected member function).

AUTHOR

Neil Moore <*neil@cs.uky.edu*>, <*nmoor0@sac.uky.edu*>

BUGS

The **Properties** class could be made more general in a number of places. For example, **load()** should take a function object argument, which would be called for each error in the input file. In addition, functions such as **readPropLine()**, **writePropLine()**, and **match_prop()** should be made virtual, so that subclasses of **Properties** could redefine them. Finally, the property list should probably be stored in a **map** (which would probably be implemented All of these improvements may be implemented in the future.