Chapter 9. Parameters and States

Authors: *Joseph T. Buck*

Other Contributors: *Neil Smyth*

A State is a data structure associated with a block, used to remember data values from one invocation to the next. For example, the gain of an automatic gain controller is a state. A state need not be dynamic; for instance, the gain of fixed amplifier is a state. A parameter is the initial value of a state. A State actually has two values: the initial value, which is always a character string, and a current value, whose type is different for each derived class of State: integer for IntState, an array of real values for FloatArrayState, etc. In addition, states have attributes, which represent logical properties the state either has or does not have.

9.1 Class State

Class State is derived from class NamedObj. The State base class is an abstract class; you cannot create a plain State. The base class contains the initial value, which is always a `const char*`; the derived classes are expected to provide current values of appropriate type. The constructor for class State sets the initial value to a null pointer, and sets the state's attributes to a value determined by the constant AB_DEFAULT, which is defined in "State.h" to be the bitwise or of AB_CONST and AB_SETTABLE. The destructor does nothing extra.

9.1.1 State public members

```
State& setState(const char* stateName, Block* parent,
               const char* initValue, const char* desc = NULL);
```

This function sets the name, parent, initial value, and optionally the descriptor for a state. The character strings representing the initial value and descriptor must outlive the State.

```
State& setState(const char* stateName, Block* parent,
               const char* initValue, const char* desc,
               Attribute attr);
```

This function is the same as the other `setState,` but it also sets attributes for the state. The Attribute object represents a set of attribute bits to turn on or off.

```
void setInitValue(const char* valueString);
```

This function sets the initial value to *`valueString.`* This string must outlive the State.

```
const char* initValue () const;
```

Return the initial value.

```
virtual const char* type() const = 0;
```

Return the type name (for use in user interfaces, for example). When states are created dynamically (by the `KnownState` or `InterpGalaxy` class), it is this name that is used to specify the type.

`virtual int size() const;`

Return the size (number of distinct values) in the state. The default implementation returns 1. Array state types will return the number of elements.

`virtual int isArray() const;`

Return TRUE if this state is an array, false otherwise. The default implementation returns false.

`virtual void initialize() = 0;`

Initialize the state. The `initialize` function for a state is responsible for parsing the initial value string and setting the current value appropriately; errors are signaled using the `Error::abortRun` mechanism.

`virtual StringList currentValue() const = 0;`

Return a string representation of the current value.

`void setCurrentValue(const char*` *`newval`*);

Modify the current value, in a type-independent way. Notice that this function is not virtual. It exploits the semantics of `initialize` to set the current value using other functions; the initial value is not modified (it is saved and restored).

`virtual State* clone() const = 0;`

Derived state classes override this method to create an identical object to the one the method is called on.

`StringList print(int` *`verbose`*) const;

Output all info. This is NOT redefined for each type of state.

`bitWord attributes() const;`

Return my attribute bits.

`bitWord setAttributes(const Attribute&` *`attr`*);
`bitWord clearAttributes(const Attribute&` *`attr`*);

Set or clear attributes.

`const State* lookup(const char*` *`name`*`, Block*` *`b`*`);`

This method searches for a state named *`name`* in Block *`b`* or one of its ancestors, and either returns it or a null pointer if not found.

`int isA(const char*) const;`

This function returns true when given the name of the class or the name of any baseclass

9.1.2 The State parser and protected members

Most of the protected interface in the State class consists of a simple recursive-descent parser for parsing integer and floating expressions that appear in the initial value string. The ParseToken class represents tokens for this parser. It contains a token type (an integer code) and a token value, which is a union that represents either a character value, a string value, an integer value, a double value, a Complex value, or a State value (for use when the initializer references another state). Token types are equal to the ASCII character value for single-character tokens. Other possible token values are:

- `T_EOF` for end of file,
- `T_ERROR` for error,
- `T_Float` for a floating value,
- `T_Int` for an integer value,
- `T_ID` for a reference to a state, and
- `T_STRING` for a string value.

For most of these, the token value holds the appropriate value. Most derived State classes use this parser to provide uniformity of syntax and error reporting; however, it is not a requirement to use it. Derived `State` classes are expected to associate a `Tokenizer` object with their initial value string. The functions provided here can then be used to parse expressions appearing in that string.

`ParseToken getParseToken(Tokenizer&` *`tok`*`, int` *`stateType`* `= T_Float);`

This function obtains the next token from the input stream associated with the Tokenizer. If there is a pushback token, that token is returned instead. If it receives a '<' token, then it assumes that the next string delimited by white space is a file name. It substitutes references to other parameters in the filename and then uses the Tokenizer's include file capability to insert the contents of the file into the input stream. If it receives a '!' token, then it assumes that that the next string delimited by white space is a command to be evaluated by an external interpreter. It substitutes references to other parameters in the command, sends the resulting string to the interpreter defined by interp member described above for evaluation, and inserts the result into the input stream. The information both read from an external file and returned from an external interpreter is also parsed by this function. Therefore, the external interpreter can perform both numeric and symbolic computations. When the parser hits the end of the input stream, it returns T_EOF.

The characters in the set `[]+*-/()^` are considered to be special and the lexical value is equal to the character value. Integer and floating values are recognized and evaluated to produce either T_Int or T_Float tokens. However, the decision is based on the value of *`stateType;`* if it is T_Float, all numeric values are returned as T_Float; if it is T_Int, all numeric values are returned as T_Int. Names that take the form of a C or C++ identifier are assumed to be names of states defined at a higher level (states belonging to the parent galaxy or some ancestor galaxy). They are searched for using `lookup;` if not found, an error is reported using `parseError` and an error token is returned. If a State is found, a token of type T_ID is returned if it is an array state or COMPLEX; otherwise the state's current value is substituted and reparsed as a token. This means, for example, that a name of an IntState will be replaced with a T_Int token with the correct value.

`void parseError (const char*` *`part1`*`, const char*` *`part2`* `= "");`

This method produces an appropriately formatted error message with the name of the state and the arguments and calls `Error::abortRun.`

`static ParseToken pushback();`
`static void setPushback(const ParseToken&);`
`static void clearPushback();`

These functions manipulate the pushback token, for use in parsing. The first function returns the current pushback token, the second sets it to be a copy of the argument, the third clears it. There is only one such token, so the state parser is not reentrant.

`ParseToken evalIntExpression(Tokenizer&` *`lexer`*`);`
`ParseToken evalIntTerm(Tokenizer&` *`lexer`*`);`
`ParseToken evalIntFactor(Tokenizer&` *`lexer`*`);`
`ParseToken evalIntAtom(Tokenizer&` *`lexer`*`);`

These four functions implement a simple recursive-descent expression parser. An expression is either a term or a series of terms with intervening '+' or '-' signs. A term is either a factor or a series of factors with interventing '*' or '/' signs. A factor is either an atom or a series of atoms with intervening '^' signs for exponentiation. (Note, C fans! ^ means exponentiation, not exclusive-or!). An atom is any number of optional unary minus signs, followed either by a parenthesized expression or a T_Int token. If any of these methods reads too far, the pushback token is used. All `getParseToken` calls use *`stateType`* T_Int, so any floating values in the expression are truncated to integer. The token types returned from each of these methods will be one of T_Int, T_EOF, or T_ERROR.

`ParseToken evalFloatExpression(Tokenizer&` *`lexer`*`);`
`ParseToken evalFloatTerm(Tokenizer&` *`lexer`*`);`
`ParseToken evalFloatFactor(Tokenizer&` *`lexer`*`);`
`ParseToken evalFloatAtom(Tokenizer&` *`lexer`*`);`

These functions have the identical structure as the corresponding Int functions. The token types returned from each of these methods will be one of T_Float, T_EOF, or T_ERROR.

`InvokeInterp interp;`

An external interpreter for evaluating commands in a parameter definition preceded by the ! character and surrounded in quotes. By default, no interpreter is defined. If the interpreter were defined as the Tcl interpreter, then ! "expr abs(cos(1.0))" would compute 0.540302. Other parameters can be referenced as usual by using curly braces, e.g. ! "expr abs(cos({gain}))".

`StringList parseFileName(const char*);`

This method parses filenames that have been inherited from state values enclosed in curly braces.

`StringList parseNestedExpression(const char*` *`expression`*`);`

This method parses nested sub-expressions appearing in the *`expression`* , e.g. {{{FilterTapFile}/{File}}}, that might be passed off to another interpreter for evaluation, e.g. Tcl.

`Int mergeFileContents(Tokenizer&` *`lexer`*`, char*` *`token`*`);`

This method treats the next token on the *`lexer`* as a filename.

`Int sendToInterpreter(Tokenizer&` *`lexer`*`, char*` *`token`*`);`

This method sends the next token on the *`lexer`* to be evaluated by an external interpreter.

`Int getParameterName(Tokenizer& lexer, char* token);`

This method looks for parameters of the form `{name}.`

9.2 Types of states

9.2.1 Class IntState and class FloatState

Class `IntState`, derived from `State`, has an integer current value. Its `initialize()` function uses the `evalIntExpression` function to read an integer expression from the initial value string. If successful, it attempts to read another token from the string; if there is another token, it reports the error “extra text after valid expression”. An assignment operator is provided that accepts an integer value and loads it into the current value. A cast to integer is also defined for accessing the current value. The virtual function `currentValue` is overloaded to return a printed version of the current value. In addition to the `setInitValue` from class State, a second form is provided that takes an integer argument. Standard overrides for `isA`, `className`, and `clone` are provided. Class `FloatState` is almost identical to class IntState except that its data field is a double precision value; where IntState functions have an argument or return value of `int`, FloatState has a corresponding argument or return value of `double`. Both are generated from the same pseudo-template files. The `type()` function for IntState returns `"INT"`. For FloatState, `"FLOAT"` is returned. For both implementations, a prototype object is added to the KnownState list.

9.2.2 Class ComplexState

ComplexState is much like FloatState and IntState, except in the expressions it accepts for initial values. Its data member is Complex and it accordingly defines an assignment operator that takes a complex value and a conversion operator that returns one. The initial value string for a ComplexState takes one of three forms: it may be the name of a galaxy ComplexState, a floating expression (of the form accepted by `State::evalFloatExpression`), or a string of the form (*`floatexp1`* , *`floatexp2`*) where both *`floatexp1`* and *`floatexp2`* are floating expressions. For the second form, the imaginary part will always be zero. For the third form, the first expression gives the real part and the second gives the imaginary part.

9.2.3 Class StringState

A StringState’s current value is a string (more correctly, of type `const char*`). The current value is created by the `initialize()` function by scanning the initial value string. This string is copied literally, except that curly braces are special. If a pair of curly braces surrounds the name of a galaxy state, the printed representation of that state’s current value (returned by the `currentValue` function) is substituted. To get a literal curly brace in the current value, prefix it with a backslash. Class StringState defines assignment operators so that different string values can be copied to the current value; the value is copied with `saveString` and deleted by the destructor.

9.2.4 Numeric array states

Classes IntArrayState and FloatArrayState are produced from the same pseudo-template. Class ComplexArrayState has a very similar design. All return `TRUE` to `isArray`, provide an array element selection operator (`operator[](int)`), and an operator that converts the state into a pointer to the first element of its data (much like arrays in C). The expression parser for FloatArrayState accepts a series of “subarray expressions”, which are concatenated together to get the current value when `initialize()` is called. Subarray expressions may specify a single element, some number of copies of a single element, or a galaxy array state of the same type (another FloatArrayState). A single element specifier may either be a floating point value, a scalar (integer or floating) galaxy state name, or a general floating expression enclosed in parentheses. A number of copies of this single element can be specified by appending an integer expression enclosed in square brackets. The expression parsers for IntArrayState and ComplexArrayState differ only that where FloatArrayState wants a floating expression, IntArrayState wants an integer expression and ComplexArrayState wants a complex expression (an expression suitable for initializing a ComplexState).

9.2.5 Class StringArrayState

As its name suggests, the current value for a StringArrayState is an array of strings. White space in the initial value string separates “words”, and Each word is assigned by `initialize()` into a separate array element. Quotes can be use to permit “words” to have white space. Current values of galaxy states can be converted into single elements of the StringArrayState value by surrounding their names with curly braces in the initial value. Galaxy StringArrayState names will be translated into a series of values. There is currently no provision for modifying the current value of a StringArrayState other than calling of `initialize` to parse the current value string.