ArguL Manifesto (DRAFT)

Vincent Hugot

February 10, 2010

1 What is ArguL?

BEWARE! this is the draft of the draft of the draft of an incomplete document.

2 Vocabulary and ArguL specification

This is a very messy section with vocabulary and ideas thrown about higgledy-piggledy.

A command line looks like this:

$$\texttt{some-program} \quad a_1 \quad a_2 \quad \ldots \quad a_n$$

The a_i are the *arguments of the program*, as provided by the underlying shell. In ArguL, an argument is either

- A *flag* (or *concrete option*). A flag is of the form `-x`, `+x`, `~x`, `++long_x`, etc... (see paragraph on flags below)
- An *option argument*, or *optarg*, that is to say anything that follows a flag whose referenced option expects arguments.
- The *killer*: `--`. This indicates that any upcoming argument must be viewed as an operand, and not a flag.
- An *operand*: Anything which follows the killer, or is neither the killer nor an option or optarg. Depending on settings, operands may need to be strictly the last arguments.

An *option* (or *abstract option*) o is a word with a given *arity* $n = \alpha(o)$. Every option has a fixed, finite arity. Options with variable arity are not allowed in **Argu**L, which means that optargs may not be optional themselves. An option o is referenced by a number of flags. If a_k is a flag for option o of arity n, then the arguments $a_{k+1}, \ldots, a_{k+n}$ will be read as optargs for o, no matter what they look like. If there are not enough arguments, then the command line is invalid.

An *extended hyphen* (or *xphen*) is one of -, +,~ (*short* xphens). A *long* xphen is a short xphen, doubled (for instance, ++). An *xphen* is *positive* if it is a short or long +, *negative* if it is a short or long -, *mutating* if it is a short or long ~.

A flag can be a *(simple) short flag*, a *multiple short flag*, a *long flag*, or an *abbreviated long flag*. In all cases, it is an argument, which is prefixed by an xphen and suffixed by a word (the *flag payload*), which is at least a character long and does not begin with - (-- is the killer, not a flag). A flag is short or long depending on whether its xphen is itself short or long. If a flag is short, and its payload is a single character, then it is a simple short flag. If its payload is longer, then it is a multiple short flag. Let $f = xc_1c_2 \ldots c_n$ be a multiple short flag, where x denotes a short xphen. Then this can be replaced by the arguments $xc_1, \ldots, xc_n$. Of course this only makes sense if all short flags c_1 to c_{n-1} reference constant options (options with arity 0). The case of a long flag is straightforward. In the case of something which has the form of a long flag but for which no corresponding long payload has been declared, we compute the set of options referenced by the declared long payloads of which the current payload is a prefix. If this set is a singleton $\{o\}$, then this is an abbreviated long flag, and is replaced by any flag which references o (this means that even if there are several possible long flags which extend this, it's alright if they reference the same abstract option anyway). In case the set is empty, then we have an *invalid long flag*. Lastly, if the set is greater than a singleton, we have an *ambiguous abbreviation*.

If a flag (of any kind) references an option o of positive arity, it is *functional*. If not, it references a *constant option* (option of arity 0) (also called *switch*), and then it is said to be *positive*, *negative* or *mutating* depending on the quality of its xphen. Since a constant option can be assimilated to a boolean variable, a positive flag will set its value to "true", a negative flag will set it to "false", and a mutating one will invert its value. This is (to my mind) an improvement over GNU style, because for instance if you have `v` for verbose mode you do `+v` to activate it, `-v` to deactivate it, `~v` to change from the default, whatever that may be. You don't need another option `q` for "quiet". How intuitive is it that `q` negates `v`? Besides, it wastes a letter which could be used for something else. Moreover, this makes it safer to change the defaults of the program: in

GNU, the same `-x` can mean "activate this" or "deactivate it", depending of whether the feature is deactivated by default or not. So on top of remembering which letter references which feature of the program, you have to remember the program's default settings in order to properly read or write the command-line. And should those defaults ever change in a new version, the meaning of your own flags will change. Not good.

Any xphen may be used for a functional flag, without consequences. A `-` is recommended, because it is the least obtrusive symbol, but it really does not matter. In some cases, the user may want to use the xphen which best reflects the intuition of what the flag does. An example is given below.

An option is *called* by the command-line is at least one flag referencing it belongs to the command-line.

Options may be grouped into categories (potentially overlapping, though there are few use cases for this case), and categories made mutually exclusive. If $C_1, \ldots, C_n$ are mutually exclusive categories of options, then if $o \in C_i$ is called by the command-line, the command-line will be invalid if any option $p \in C_j$, $j \neq i, p \neq o$, is also called. For convenience, the user can define the categories and express the exclusion conditions on unions,intersections and set differences of the categories she defined. Several exclusion conditions can be defined, and must be met simultaneously in order for the command-line to be correct.

The user may state *call bounds* for an option o; for instance stating that is must be called at least n times, and/or at most m times. Failing that the command-line will be invalid

For any optarg, the user may specify simple checks: for instance verify that it codes an integer, with optional bounds. The user may implement hooks to perform more complex tasks.

When the command-line is parsed, the user must have access to the values and number of calls of each options, from their option name. Optionally, the implementation may provide a map from optarg name to value.

Note: The number of calls of an option may be used to carry more information on a switch than just its value: for instance `+vvv` could code verbosity level 3. Some GNU programs use this kind of voodoo. It is a possibility in **Argu**L, but definitely not recommended. It would be cleaner to use an unary option `+v 3`, to set to lvl 3, and `-v 0` to deactivate. (Note the use of + and -: any xphen would have been valid since the flag is not functional, but they were used to reflect the idea of activating and deactivating verbose mode, and improve legibility of the command-line)

3 The language

draft by example:

```
argul ‘‘program name’’

preamble ‘‘some code’’

options
// list of options
// ! precedes any identifier we define so as to
// prevent conflict of user-defined names and
// language keywords.
// other possibility: clause keywords
// switch to other lexing context. More elegant but
// more expensive.
  !verbose [false]:// option name and name optargs: here a switch.
    // default value is false: syntax just for switches
    short v,V;  // short flags (clause 1)
    long !verbose;  // long flags ...
      // if no long clause, the default is to take the option name
      // here the given clause is equivalent to no clause.
      // it is possible to specify an empty clause: long;
    cat !verbosity; // categories
    . // end option. Clauses are optional and
      // can appear in any order, but at most once.

  !output <!filename:!file=’’’’>: // an unary option
    // has one argument, which must pass ‘‘file’’ test
    // built-in test: function string->bool
    // true iff file exists
    // built-in tests are file,int,intRange x y, where
    // x,y integers or Top/Bottom
    // also float and floatRange.
    // third component is default value.
    // both test and default value are optional
    short o;
    // long !output; is implied...
    doc 1 ‘‘indicate output file’’;
```

```
    doc 2 ‘‘some more doc’’;
    // documentation clauses, with cumulative levels.
    // for instance the help page will use lvl 1 only,
    // while a man page would use lvl 1 as a short desc,
    // and also lvl 2 as a more in-depth description
    cat !outputs;
    occurrences n >= 1; // occurs at least once. n is always
      // the total number of occurences
    .

  !seven_ary <!an_int:!int> <!alist:!file>*6:
    // expects an int and 6 files
    // stupid but illustates the grammar
    cat !outputs;
    // note: long seven-ary; is implied.
    // the _ becomes -.
    .

macros
  macro:
    short m;
    long !my-macro;
    is ‘‘-v’’,‘‘-o’’;
    doc 1 ‘‘activates verbose mode and behaves as -o’’;
    // macros are replaced by a series of other
    // hard-coded command-line arguments
    .

exclusions
  excl verbosity,outputs;
  // if we had defined the categories A,B,C,...
  excl (A or B) and C, D;
  // if you have an option that's in C and in either A or B,
  // then you can't use one in D. And vice versa.
  // not sure this is useful.
  .

postamble ‘‘some code’’
```