NAME

arg_parse – parse arguments to a command

SYNOPSIS

#include <arg.h>
int **arg_parse**(int argc, char **argv, [char *formatstr, paramptrs, char *docstr, docparams]*, 0)

double **expr_eval**(char *str)

Arg_form *__arg_to_form__(0, [char *formatstr, paramptrs, char *docstr, docparams]*, 0)

DESCRIPTION

arg_parse is a subroutine for parsing and conversion of command-line arguments. It can be called from ANSI C or C++. This parser is an alternative to the common method of argument parsing, which is an ad-hoc parser in each program, typically written with a large, cumbersome switch statement. *arg_parse* allows a command-line parser to be described very concisely while retaining the flexibility to handle a variety of syntaxes.

The parser has a number of features:

- arbitrary order of flag arguments
- automatic argument conversion and type checking
- multiple-character flag names
- required, optional, and flag arguments
- automatic usage message
- subroutine call for exotic options (variable number of parameters)
- modularized parsers encourage standardized options
- expression evaluation
- works either from argv or in interactive mode, as a primitive language parser and interpreter
- concise specification
- easy to use

It is hoped that use of *arg_parse* will help standardize argument conventions and reduce the tedium of adding options to programs.

APPETIZER

Here is a simple example:

```
#include <arg.h>

void main(int argc, char **argv) {
    char *file;
    int level = 3, debug;
    double xsize = 20., ysize = 10.;

    arg_parse(argc, argv,
            "", "Usage: prog [options]",
            "%S", &file, "set output file",
            "[%d]", &level, "set recursion level [default=%d]", level,
            "-size %F %F", &xsize, &ysize, "set x and y sizes",
            "-debug", ARG_FLAG(&debug), "turn on debugging",
    0);
    ...
```

The *arg_parse* call defines the program's arguments, in this case: one required argument (a filename),

an optional argument (an integer level number), an optional flag with two parameters (floating point size), and a simple flag (boolean debug flag). If the above program (call it *prog*) were run with

```
prog joe.c
```

it would set *file* to joe.c, and set *debug* to 0, and if run with

```
prog -size 100 400/3 joe.c -debug 5
```

it would set *file*="joe.c", *level*=5, *xsize*=100, *ysize*=133.33, and *debug*=1. In all programs using *arg_parse*, a hyphen argument elicits a usage message, so the command

```
prog -
```

results in the printout

```
Usage: prog [options]
%S              set output file
[%d]            set recursion level [default=3]
-size %F %F     set x and y sizes
-debug          turn on debugging
```

TERMINOLOGY

In order to speak precisely about the description and use of argument parsers, it helps to define some terminology.

TERM	EXAMPLES	MEANING
argument	-size joe.c	Any of the strings in argv, supplied by the user.
flag arg	-size	The name of an option.
parameter arg	100	A value (numerical or otherwise) for an option.
simple flag	-debug	A flag with no parameters that sets a boolean variable.
regular arg	joe.c	An argument that is not a flag or a parameter to a flag. Can be either a required or optional argument.
format string	"-size %F%F"	The character string describing the syntax of an option.
parameter ptr	&xsize	Pointer to a parameter variable through which converted values are stored.
doc string	"set output file"	Documentation string describing the option's effect.
form	"-res%d", &r, "set res" "[%d]", &level, "set level"	Format string, parameter pointers, and documentation describing an option.

We will describe the syntax of formlists first, then the method for matching arguments to forms.

FORMLIST SYNTAX

The syntax and conversion rules for parsing are specified in the **formlist** following *argc* and *argv* in the *arg_parse* call. *arg_parse* reads its subroutine parameters using the ANSI C *stdarg(5)* convention for procedures with a variable number of arguments, so it is crucial that the formlist be terminated with a

0. Each form consists of a *scanf*-style format string, a list of parameter pointers, a documentation string, and a list of documentation parameters. In some cases the paramptr and docparam lists will be empty, but the format string and doc string arguments are mandatory.

Format String

The format string consists of a flag string followed by parameter conversion codes (if any). A flag is a hyphen followed by a string. None of the characters in the string may be a '%' and the string must not begin with a numeral. Acceptable conversion codes in the format string are a '%' followed by any single character codes accepted by *scanf* plus the new conversion 'S':

CODE	TYPE
%c	char
%d	int
%f	float
%F	double
%s	char array
%S	char *
...	(see *scanf(3)* for a complete list)

The %S conversion is like %s except it copies only a pointer to a string (a char *), not a whole string. When using %s, space must be allocated for the copied string, but with %S only room for a pointer is needed. An example of %S use is given later. A format string with no flag but only conversion codes describes a **regular argument**, while a flag followed by conversion codes defines a **flag with arguments**. Brackets around conversion codes indicate that they are optional, for example:

"%S %d"	two required args
"%d [%F]"	first arg required, second arg optional
"-pt [%F%F%F[%F]]"	a flag with 0, 3, or 4 parameters

Since assignments of args to parameter pointers are done left-right within the form, no conversion codes can follow the first ']'. In fact, the]'s are optional since they can be inferred to be at the end of the format string. Spaces between conversion codes are optional and ignored.

Following the format string is the list of parameter pointers, whose number must match the number of conversion codes in the format string, like the arguments to *scanf* or *printf*.

Form Types

There are six form types. In addition to the ones we've seen, regular arguments and flags with parameters, there are several others for more exotic circumstances: simple flags, nop forms, subroutine flags, and sublists.

A **simple flag** is a flag option with no parameters that sets a boolean variable to 1 if that flag appears in *argv*, else 0. A pointer to the boolean (int) variable is passed after the format string using the ARG_FLAG macro. For example, ARG_FLAG(&debug) will set the boolean variable debug.

A **nop form** is a documentation string with no associated flags or arguments that appears in the usage message but does not affect parsing. Nop forms have a format string and a doc string, the former containing neither a flag nor a conversion code. Example:

```
"", "This program converts an AIS picture file to PF format",
```

When the usage message is printed, the doc string is indented if the format string is non-null.

A **subroutine flag** is an option that calls a user-supplied *action subroutine* every time it is used rather than using *arg_parse*'s format conversion and parameter assignment. Subroutine flags are used just like flags with parameters in *argv*, but they are specified and implemented differently internally. For example, say our program *prog* needs a variable length list of people. We could add a flag with arguments

to handle a few names using the form:

```
char *p1, *p2, *p3, *p4;
...
"-people %S[%S[%S[%S]]]]", &p1, &p2, &p3, &p4, "people names"
```

but this limits the number of possible parameters to four. Subroutine flags provide a trapdoor whereby the programmer can do custom conversion or processing of parameters with arbitrary type and number. To parse our list of people with a subroutine flag instead, we use the form:

```
"-people", ARG_SUBR(arg_people), "people names"
```

where arg_people is a subroutine to gobble the parameters, just like in the example near the end of this document.

The macro ARG_SUBR takes the name of a subroutine to call when the flag is encountered. The subroutine is called with a new argument count *ac* and a new argument vector *av*. The latter is formed by re-packaging the parameter arguments from *argv* that follow the flag. In our list-of-people example, the command prog foo -people ned alvy bruce -debug *would call arg_people* with *ac*=3 and *av*={"ned","alvy","bruce"}.

Whereas flags with arguments had the simple side effect of setting a variable, subroutine flags can have arbitrarily complex side effects, and can be used multiple times. Subroutine flags can also be flagless; that is, they can have null format strings. In this case, any ''leftover'' regular arguments are passed to the supplied action subroutine. Flagless subroutines are useful for reading lists of filenames.

The final form type is a **sublist**. A sublist is a subordinate parser defined as another formlist. Sublists can be used to build a tree of parsers, for example a 3-D graphics program might have a standard set of commands for controlling the display (setting the output device, screen window, and colors) and also a standard set of commands for transforming 3-D objects (rotation, scaling, etc.). Within the display command parser there could well be a standard set of commands for each output device (one for Suns, another for Versatec plotters, etc.). Using sublists we can prepare a standard parser for display commands and keep it in the source for the display library, a parser for the transformation commands in the transformation library, and so on, so that the parser for each graphics application can be very simple, merely listing its own options and then invoking the standard parsers for the major libraries it uses to handle the bulk of the options. Modularizing parsers in this way reduces the redundancy of parsing code between similar commands and encourages standardization of options between programs, reducing maintenance work for programmers and reducing option confusion among users.

To invoke a sublist we use the form:

```
"-display", ARG_SUBLIST(form), "display commands"
```

The ARG_SUBLIST macro expects a structure pointer of type Arg_form * as returned from the arg_to_form routine. arg_to_form requires arguments identical to arg_parse except in place of argc,argv one passes a 0. Its use is illustrated in an example later.

MATCHING ARGUMENTS TO FORMS

arg_parse steps through the arguments in *argv* from left to right, matching arguments against the format strings in the formlist. Flag arguments (simple flags or flags with parameters) can occur in arbitrary order but regular arguments are matched by stepping through the formlist in left to right order. For this reason regular arguments are also known as positional arguments. Matching of parameters within an option is also done in a left-to-right, greedy fashion within the form without regard for the parameter types. No permutation of the matching is done to avoid conversion errors. To illustrate, in our *prog* above, if we changed the size option to make the second parameter optional:

```
"-size %F[%F]", &xsize, &ysize, "set sizes",
```

then the command:

```
prog -size 100 -debug joe.c
```

succeeds because it is clear that only one parameter is being supplied to size, but if we try:

```
prog -size 100 joe.c -debug
```

then *arg_parse* will attempt to convert "joe.c" *via %F* into *ysize* and fail, returning an error code.

The matching algorithm for subroutine flags and sublists varies somewhat from that for the other form types. For most types, *arg_parse* grabs as many arguments out of *argv* as the form can take up to the next flag argument (or the end of *argv*), but for subroutine flags and sublists, all arguments up to the next flag argument are grabbed and bundled into a smaller argument vector (call it *av*). (For matching purposes, a flag argument is an argument that begins with a hyphen followed by any character except digits and '.'.) The new argument vector is passed to the action routine in the case of subroutine flags or recursively to a sub-parser in the case of sublist flags.

The sub-parser invoked by a sublist flag does matching identically. Normally the entire formlist tree is traversed depth-first whenever a search for a flag is being made. If there are no flag duplicates between different levels of the form tree then the structure of the tree is irrelevant; the user needn't be conscious of the command grouping or of the sublist names. But if there are name duplicates, for example if there were a -window option in both the display and transformation parsers, then explicit control of search order within the tree is needed. This disambiguation problem is analogous to pathname specification of files within a UNIX directory tree. When explicit sublist selection is needed it is done using the sublist flag followed by the arguments for the sub-parser, bracketed with -{ and -} flags. For example, if there were more than one window option, to explicitly select the one in the display parser, we type:

```
-display -{ -window 0 0 639 479 -}
```

The brace flags group and quote the arguments so that all of the enclosed arguments will be passed to the sub-parser. Without them the argument matcher would think that display has no parameters, since it is immediately followed by a flag (-window). Note that in *csh*, the braces must be escaped as -\{ *and* -\}.

[If you can think of a better way to do matching please tell me! -Paul].

The matching is checked in both directions: in the formlist, all required arguments must be assigned to and most flags can be called at most once, and in *argv*, each argument must be recognized. Regular arguments are **required** if they are unbracketed, and **optional** if they are bracketed. Unmatched forms for required arguments cause an error but unmatched forms for optional or flag arguments do not; they are skipped. A warning message is printed if a simple flag or flag with parameters appears more than once in *argv*. Note that it is not an error for subroutine flags to appear more than once, so they should be used when repeats of a flag are allowed. Unmatched arguments in *argv* cause an ''extra argument'' error.

A hyphen argument in *argv* causes *arg_parse* to print a usage message constructed from the format and documentation strings, and return an error code.

EXPRESSIONS

arg_parse does expression evaluation when converting numerical parameters. The expression evaluator allows the following operations: +, -, *, /, % (mod), ^ (exponentiation), unary -, unary +, *sqrt*, *exp*, *log*, *pow*, *sin*, *cos*, *tan*, *asin*, *acos*, *atan*, *atan2* (takes 2 args), *sind*, *cosd*, *tand*, *dasin*, *dacos*, *datan*, *datan2* (takes 2 args), *floor*, and *ceil*. It also knows the two constants *pi* and *e*. Numerical constants can be integer or scientific notation, in decimal, octal, hexidecimal, or other base. For example, 10 = 012 (base 8) = 0xa (base 16) = 0b2:1010 (base 2). The normal trig functions work in radians, while the versions that begin or end in the letter 'd' work in degrees. Thus, "exp(-.5*2^2)/sqrt(2*pi)" is a legal

expression. All expressions are computed in double-precision floating point. Note that it is often necessary to quote expressions so the shell won't get excited about asterisks and parentheses. The expression evaluator *expr_eval* can be used independently of *arg_parse*.

INTERACTIVE MODE

If the lone argument -stdin is passed in *argv* then *arg_parse* goes into interactive mode. Interactive mode reads its arguments from standard input rather than getting them from the argument vector. This allows programs to be run semi-interactively. To encourage interactive use of a program, one or more of the options should be a subroutine flag. One could have a -go flag, say, that causes computation to commence. In interactive mode the hyphens on flags are optional at the beginning of each line, so the input syntax resembles a programming language. In fact, scripts of such commands are often saved in files.

EXAMPLE

The following example illustrates most of the features of *arg_parse*.

```
/* tb.c - arg_parse test program */
#include <stdio.h>
#include <stdlib.h>

#include <arg.h>
static double dxs = 1., dys = .75;
static int x1 = 0, y1 = 0, x2 = 99, y2 = 99;
static char *chanlist = "rgba";

static void arg_people(int argc, char **argv) {
    int i;

    for (i=0; i<argc; i++)
            printf("person[%d]=%s\n", i, argv[i]);
}

static void arg_dsize(int argc, char **argv) {
    if (argc<1 | argc>3) {
            fprintf(stderr, "-dsize wants 1 or 2 args\n");
            exit(1);
    }
    /* illustrate two methods for argument conversion */
    dxs = atof(argv[0]);                                    /* constant conversion */
    if (argc>1) dys = expr_eval(argv[1]);     /* expression conversion */
    else dys = .75*dxs;
}

Arg_form *fb_init() {
    return arg_to_form(0,
            "-w%d%d%d%d", &x1, &y1, &x2, &y2, "set screen window",
            "-ch%S", &chanlist, "set channels [default=%s]", chanlist,
    0);
}

void main(int argc, char **argv) {
    int fast, xs = 512, ys = 486;
    double scale = 1.;
    char *fromfile, tofile[80], *child = "jim";
    Arg_form *arg_fb;

    arg_fb = fb_init();
```

```
    if (arg_parse(argc, argv,
            "", "Usage: %s [options]", argv[0],
            "", "This program does nothing but test arg_parse",
            "%S %s", &fromfile, tofile, "fromfile and tofile",
            "[%F]", &scale, "set scale [default=%g]", scale,
            "", ARG_SUBR(arg_people), "names of people",
            "-fast", ARG_FLAG(&fast), "do it faster",
            "-ch %S", &child, "set child name",
            "-srcsize %d[%d]", &xs, &ys, "set source size [default=%d,%d]", xs, ys,
            "-dstsize", ARG_SUBR(arg_dsize), "set dest size",
            "-fb", ARG_SUBLIST(arg_fb), "FB COMMANDS",
    0) < 0)
            exit(1);

    printf("from=%s to=%s scale=%g fast=%d child=%s src=%dx%d dst=%gx%g\n",
            fromfile, tofile, scale, fast, child, xs, ys, dxs, dys);
    printf("window={%d,%d,%d,%d} chan=%s\n", x1, y1, x2, y2, chanlist);
}
```

In this example we have two required arguments, one optional argument, and a flagless subroutine (arg_people) to gobble the remaining regular arguments. The two required arguments illustrate the differences between %S and %s, and the advantages of the former. The -srcsize and -dstsize forms illustrate two different ways to get a flag with either one or two parameters. Note in the *arg_dsize* routine that the expression evaluator *expr_eval* is just as easy to use as *atof*. A small sublist shows an example of command name ambiguity in the flag -ch.

Below are the results of several sample runs.

- tb one two
  from=one to=two scale=1 fast=0 child=jim src=512x486 dst=1x0.75
  window={0,0,99,99} chan=rgba

Only the two required args are specified here and everything else defaults.

- tb -fast -srcsize 100 1+2 one two -dstsize 2 -ch amy -w 1 2 3 4 "sqrt(2)"
  from=one to=two scale=1.41421 fast=1 child=amy src=100x3 dst=2x1.5
  window={1,2,3,4} chan=rgba

This illustrates expression evaluation, the precedence of the first -ch *flag over the one in the sublist, and easy access to a non-ambiguous sublist option,* -w.

- tb -fb -\{ -ch abc -w 9 8 7 6 -\} -ch -\{ -jo -\} A B 44 larry curly moe
  person[0]=larry
  person[1]=curly
  person[2]=moe
  from=A to=B scale=44 fast=0 child=-jo src=512x486 dst=1x0.75
  window={9,8,7,6} chan=abc

This shows access to a ''shadowed'' sublist option, -ch, *and escaping a parameter string that happens to begin with a hyphen,* -jo, *with braces, plus the use of a flagless subroutine to pick up extra regular arguments.*

RETURN VALUE

arg_parse returns a negative code on error, otherwise 0. The file *arg.h* contains definitions for the error codes:

ARG_BADCALL	programmer error, bad formlist
ARG_BADARG	bad argument in *argv*
ARG_MISSING	required argument or parameter to flag missing
ARG_EXTRA	*argv* contains an extra, unrecognizable argument

NOTE

arg_parse modifies *argv* as a side-effect to eliminate the -{ and -} arguments.

COMPILING

If *arg_parse* is installed in *libarg.a*, compile with cc ... -larg -lm.

SEE ALSO

scanf(3), stdarg(5)

AUTHOR

Paul Heckbert, ph@cs.cmu.edu, April 1988, Oct. 1998