

NAME

gvpr – graph pattern scanning and processing language
(previously known as **gpr**)

SYNOPSIS

gvpr [-icV?] [-o *outfile*] [-a *args*] ['*prog*' | -f *progfile*] [*files*]

DESCRIPTION

gvpr is a graph stream editor inspired by **awk**. It copies input graphs to its output, possibly transforming their structure and attributes, creating new graphs, or printing arbitrary information. The graph model is that provided by *libagraph*(3). In particular, **gvpr** reads and writes graphs using the dot language.

Basically, **gvpr** traverses each input graph, denoted by **\$G**, visiting each node and edge, matching it with the predicate-action rules supplied in the input program. The rules are evaluated in order. For each predicate evaluating to true, the corresponding action is performed. During the traversal, the current node or edge being visited is denoted by **\$**.

For each input graph, there is a target subgraph, denoted by **\$T**, initially empty and used to accumulate chosen entities, and an output graph, **\$O**, used for final processing and then written to output. By default, the output graph is the target graph. The output graph can be set in the program or, in a limited sense, on the command line.

OPTIONS

The following options are supported:

- a *args* The string *args* is split into whitespace-separated tokens, with the individual tokens available as strings in the **gvpr** program as **ARGV[0],...,ARGV[ARGC-1]**.
- c Use the source graph as the output graph.
- i Derive the node-induced subgraph extension of the output graph in the context of its root graph.
- o *outfile*
Causes the output stream to be written to the specified file; by default, output is written to **stdout**.
- f *progfile*
Use the contents of the specified file as the program to execute on the input. If *progfile* contains a slash character, the name is taken as the pathname of the file. Otherwise, **gvpr** will use the directories specified in the environment variable **GPRPATH** to look for the file. If -f is not given, **gvpr** will use the first non-option argument as the program.
- V Causes the program to print version information and exit.
- ? Causes the program to print usage information and exit.

OPERANDS

The following operand is supported:

- files* Names of files containing 1 or more graphs in the dot language. If no -f option is given, the first name is removed from the list and used as the input program. If the list of files is empty, **stdin** will be used.

PROGRAMS

A **gvpr** program consists of a list of predicate-action clauses, having one of the forms:

```
BEGIN { action }  
BEG_G { action }  
N [ predicate ] { action }  
E [ predicate ] { action }  
END_G { action }  
END { action }
```

A program can contain at most one of each of the **BEGIN**, **BEG_G**, **END_G** and **END** clauses. There can

be any number of **N** and **E** statements, the first applied to nodes, the second to edges. The top-level semantics of a **gvpr** program are: Evaluate the **BEGIN** clause, if any. For each input graph *G* {

Set *G* as the current graph and current object.

Evaluate the **BEG_G** clause, if any.

For each node and edge in *G* {

Set the node or edge as the current object.

Evaluate the **N** or **E** clauses, as appropriate.

}

Set *G* as the current object.

Evaluate the **END_G** clause, if any. } Evaluate the **END** clause, if any. The actions of the **BEGIN**, **BEG_G**, **END_G** and **END** clauses are performed when the clauses are evaluated. For **N** or **E** clauses, either the predicate or action may be omitted. If there is no predicate with an action, the action is performed on every node or edge, as appropriate. If there is no action and the predicate evaluates to true, the associated node or edge is added to the target graph.

Predicates and actions are sequences of statements in the C dialect supported by the *libexpr(3)* library. The only difference between predicates and actions is that the former must have a type that may interpreted as either true or false. Here the usual C convention is followed, in which a non-zero value is considered true. This would include non-empty strings and non-empty references to nodes, edges, etc. However, if a string can be converted to an integer, this value is used.

In addition to the usual C base types (void, int, char, float, long, unsigned and double), **gvpr** provides string as a synonym for char*, and the graph-based types node_t, edge_t, graph_t and obj_t. The obj_t type can be viewed as a supertype of the other 3 concrete types; the correct base type is maintained dynamically. Besides these base types, the only other supported type expressions are (associative) arrays.

Constants follow C syntax, but strings may be quoted with either "... " or '?...?'. In certain contexts, string values are interpreted as patterns for the purpose of regular expression matching. Patterns use *ksh(1)* file match pattern syntax. **gvpr** uses C++ comments.

A statement can be a declaration of a function, a variable or an array, or an executable statement. For declarations, there is a single scope. Array declarations have the form:

```
type array [ var ]
```

where the *var* is optional. As in C, variables and arrays must be declared. In particular, an undeclared variable will be interpreted as the name of an attribute of a node, edge or graph, depending on the context.

Executable statements can be one of the following:

```
{ [ statement ... ] }  
expression // commonly var = expression  
if( expression ) statement [ else statement ]  
for( expression ; expression ; expression ) statement  
for( array [ var ] ) statement  
while( expression ) statement  
switch( expression ) case statements  
break [ expression ]  
continue [ expression ]  
return [ expression ]
```

In the second form of the **for** statement, the variable *var* is set to each value used as an index in the specified array and then the associated *statement* is evaluated. Function definitions can only appear in the **BEGIN** clause.

Expressions include the usual C expressions. String comparisons using == and != treat the right hand operand as a pattern. **gvpr** will attempt to use an expression as a string or numeric value as appropriate.

Expressions of graphical type (i.e., graph_t, node_t, edge_t, obj_t) may be followed by a field reference in the form of *.name*. The resulting value is the value of the attribute named *name* of the given object. In addition, in certain contexts an undeclared, unmodified identifier is taken to be an attribute name. Specifically,

such identifiers denote attributes of the current node or edge, respectively, in **N** and **E** clauses, and the current graph in **BEG_G** and **END_G** clauses.

As usual in the *libagraph*(3) model, attributes are string-valued. In addition, **gvpr** supports certain pseudo-attributes of graph objects, not necessarily string-valued. These reflect intrinsic properties of the graph objects and cannot be set by the user.

head : node_t

the head of an edge.

tail : node_t

the tail of an edge.

name : string

the name of an edge, node or graph. The name of an edge has the form "<tail-name><edge-op><head-name>[<key>]", where <edge-op> is "->" or "--" depending on whether the graph is directed or not. The bracket part [<key>] only appears if the edge has a non-trivial key.

indegree : int

the indegree of a node.

outdegree : int

the outdegree of a node.

degree : int

the degree of a node.

root : graph_t

the root graph of an object. The root of a root graph is itself.

parent : graph_t

the parent graph of a subgraph. The parent of a root graph is **NULL**

n_edges : int

the number of edges in the graph

n_nodes : int

the number of nodes in the graph

directed : int

true (non-zero) if the graph is directed

strict : int

true (non-zero) if the graph is strict

BUILT-IN FUNCTIONS

The following functions are built into **gvpr**. Those functions returning references to graph objects return **NULL** in case of failure.

Graphs and subgraph

graph(*s* : string, *t* : string) : graph_t

creates a graph whose name is *s* and whose type is specified by the string *t*. Ignoring case, the characters U, D, S, N have the interpretation undirected, directed, strict, and non-strict, respectively. If *t* is empty, a directed, non-strict graph is generated.

subg(*g* : graph_t, *s* : string) : graph_t

creates a subgraph in graph *g* with name *s*. If the subgraph already exists, it is returned.

isSubg(*g* : graph_t, *s* : string) : graph_t

returns the subgraph in graph *g* with name *s*, if it exists, or **NULL** otherwise.

fstsubg(*g* : graph_t) : graph_t

returns the first subgraph in graph *g*, or **NULL** if none exists.

nxtsubg(*sg* : **graph_t**) : **graph_t**
returns the next subgraph after *sg*, or **NULL**.

isDirect(*g* : **graph_t**) : **int**
returns true if and only if *g* is directed.

isStrict(*g* : **graph_t**) : **int**
returns true if and only if *g* is strict.

nNodes(*g* : **graph_t**) : **int**
returns the number of nodes in *g*.

nEdges(*g* : **graph_t**) : **int**
returns the number of edges in *g*.

Nodes

node(*sg* : **graph_t**, *s* : **string**) : **node_t**
creates a node in graph *g* of name *s*. If such a node already exists, it is returned.

subnode(*sg* : **graph_t**, *n* : **node_t**) : **node_t**
inserts the node *n* into the subgraph *g*. Returns the node.

fstnode(*g* : **graph_t**) : **node_t**
returns the first node in graph *g*, or **NULL** if none exists.

nxtnode(*n* : **node_t**) : **node_t**
returns the next node after *n*, or **NULL**.

isNode(*sg* : **graph_t**, *s* : **string**) : **node_t**
looks for a node in graph *g* of name *s*. If such a node exists, it is returned. Otherwise, **NULL** is returned.

Edges

edge(*t* : **node_t**, *h* : **node_t**, *s* : **string**) : **edge_t**
creates an edge with tail node *t*, head node *h* and name *s*. If the graph is undirected, the distinction between head and tail nodes is unimportant. If such an edge already exists, it is returned.

subedge(*g* : **graph_t**, *e* : **edge_t**) : **edge_t**
inserts the edge *e* into the subgraph *g*. Returns the edge.

isEdge(*t* : **node_t**, *h* : **node_t**, *s* : **string**) : **edge_t**
looks for an edge with tail node *t*, head node *h* and name *s*. If the graph is undirected, the distinction between head and tail nodes is unimportant. If such an edge exists, it is returned. Otherwise, **NULL** is returned.

fstout(*n* : **node_t**) : **edge_t**
returns the first out edge of node *n*.

nxtout(*e* : **edge_t**) : **edge_t**
returns the next out edge after *e*.

fstin(*n* : **node_t**) : **edge_t**
returns the first in edge of node *n*.

nxtin(*e* : **edge_t**) : **edge_t**
returns the next in edge after *e*.

fstedge(*n* : **node_t**) : **edge_t**
returns the first edge of node *n*.

nxtedge(*e* : **edge_t**, *n* : **node_t**) : **edge_t**
returns the next edge after *e*.

Graph I/O

write(g : graph_t) : void

prints *g* in dot format onto the output stream.

writeG(g : graph_t, fname : string) : void

prints *g* in dot format into the file *fname*.

fwriteG(g : graph_t, fd : int) : void

prints *g* in dot format onto the open stream denoted by the integer *fd*.

readG(fname : string) : graph_t

returns a graph read from the file *fname*. The graph should be in dot format. If no graph can be read, **NULL** is returned.

freadG(fd : int) : graph_t

returns the next graph read from the open stream *fd*. Returns **NULL** at end of file.

Graph miscellany

delete(g : graph_t, x : obj_t) : void

deletes object *x* from graph *g*. If *g* is **NULL**, the function uses the root graph of *x*. If *x* is a graph or subgraph, it is closed unless *x* is locked.

isIn(g : graph_t, x : obj_t) : int

returns true if *x* is in subgraph *g*. If *x* is a graph, this indicates that *g* is the immediate parent graph of *x*.

clone(g : graph_t, x : obj_t) : obj_t

creates a clone of object *x* in graph *g*. In particular, the new object has the same name/value attributes and structure as the original object. If an object with the same key as *x* already exists, its attributes are overlaid by those of *x* and the object is returned. If an edge is cloned, both endpoints are implicitly cloned. If a graph is cloned, all nodes, edges and subgraphs are implicitly cloned. If *x* is a graph, *g* may be **NULL**, in which case the cloned object will be a new root graph.

copy(g : graph_t, x : obj_t) : obj_t

creates a copy of object *x* in graph *g*, where the new object has the same name/value attributes as the original object. If an object with the same key as *x* already exists, its attributes are overlaid by those of *x* and the object is returned. Note that this is a shallow copy. If *x* is a graph, none of its nodes, edges or subgraphs are copied into the new graph. If *x* is an edge, the endpoints are created if necessary, but they are not cloned. If *x* is a graph, *g* may be **NULL**, in which case the cloned object will be a new root graph.

copyA(src : obj_t, tgt : obj_t) : int

copies the attributes of object *src* to object *tgt*, overwriting any attribute values *tgt* may initially have.

induce(g : graph_t) : void

extends *g* to its node-induced subgraph extension in its root graph.

compOf(g : graph_t, n : node_t) : graph_t

returns the connected component of the graph *g* containing node *n*, as a subgraph of *g*. The subgraph only contains the nodes. One can use *induce* to add the edges. The function fails and returns **NULL** if *n* is not in *g*. Connectivity is based on the underlying undirected graph of *g*.

lock(g : graph_t, v : int) : int

implements graph locking on root graphs. If the integer *v* is positive, the graph is set so that future calls to **delete** have no immediate effect. If *v* is zero, the graph is unlocked. If there has been a call to delete the graph while it was locked, the graph is closed. If *v* is negative, nothing is done. In all cases, the previous lock value is returned.

Strings

sprintf(fmt : string, ...) : string

returns the string resulting from formatting the values of the expressions occurring after *fmt* according to the *printf(3)* format *fmt*

gsub(*str* : string, *pat* : string) : string

gsub(*str* : string, *pat* : string, *repl* : string) : string

returns *str* with all substrings matching *pat* deleted or replaced by *repl*, respectively.

sub(*str* : string, *pat* : string) : string

sub(*str* : string, *pat* : string, *repl* : string) : string

returns *str* with the leftmost substring matching *pat* deleted or replaced by *repl*, respectively. The characters '^' and '\$' may be used at the beginning and end, respectively, of *pat* to anchor the pattern to the beginning or end of *str*.

substr(*str* : string, *idx* : int) : string

substr(*str* : string, *idx* : int, *len* : int) : string

returns the substring of *str* starting at position *idx* to the end of the string or of length *len*, respectively. Indexing starts at 0. If *idx* is negative or *idx* is greater than the length of *str*, a fatal error occurs. Similarly, in the second case, if *len* is negative or *idx* + *len* is greater than the length of *str*, a fatal error occurs.

length(*s* : string) : int

returns the length of the string *s*.

index(*s* : string, *t* : string) : int

returns the index of the character in string *s* where the leftmost copy of string *t* can be found, or -1 if *t* is not a substring of *s*.

match(*s* : string, *p* : string) : int

returns the index of the character in string *s* where the leftmost match of pattern *p* can be found, or -1 if no substring of *s* matches *p*.

canon(*s* : string) : string

returns a version of *s* appropriate to be used as an identifier in a dot file.

xOf(*s* : string) : string

returns the string "x" if *s* has the form "x,y", where both *x* and *y* are numeric.

yOf(*s* : string) : string

returns the string "y" if *s* has the form "x,y", where both *x* and *y* are numeric.

lOf(*s* : string) : string

returns the string "lx,lly" if *s* has the form "lx,lly,urx,ury", where all of *lx*, *lly*, *urx*, and *ury* are numeric.

urOf(*s*)

urOf(*s* : string) : string returns the string "urx,ury" if *s* has the form "lx,lly,urx,ury", where all of *lx*, *lly*, *urx*, and *ury* are numeric.

sscanf(*s* : string, *fmt* : string, ...) : int

scans the string *s*, extracting values according to the *scanf*(3) format *fmt*. The values are stored in the addresses following *fmt*, addresses having the form **&***v*, where *v* is some declared variable of the correct type. Returns the number of items successfully scanned.

I/O

print(...) : void

print(*expr*, ...) prints a string representation of each argument in turn onto **stdout**, followed by a newline.

printf(*fmt* : string, ...) : int

printf(*fd* : int, *fmt* : string, ...) : int

prints the string resulting from formatting the values of the expressions following *fmt* according to the *printf*(3) format *fmt*. Returns 0 on success. By default, it prints on **stdout**. If the optional integer *fd* is given, output is written on the open stream associated with *fd*.

scanf(*fmt* : string, ...) : int

scanf(*fd* : int, *fmt* : string, ...) : int

scans in values from an input stream according to the *scanf*(3) format *fmt*. The values are stored in the addresses following *fmt*, addresses having the form **&***v*, where *v* is some declared variable of the correct type. By default, it reads from **stdin**. If the optional integer *fd* is given, input is read from the open stream associated with *fd*. Returns the number of items successfully scanned.

openF(*s* : string, *t* : string) : int

opens the file *s* as an I/O stream. The string argument *t* specifies how the file is opened. The arguments are the same as for the C function *open*(3). It returns an integer denoting the stream, or -1 on error.

As usual, streams 0, 1 and 2 are already open as **stdin**, **stdout**, and **stderr**, respectively. Since **gvpr** may use **stdin** to read the input graphs, the user should avoid using this stream.

closeF(*fd* : int) : int

closes the open stream denoted by the integer *fd*. Streams 0, 1 and 2 cannot be closed. Returns 0 on success.

readL(*fd* : int) : string

returns the next line read from the input stream *fd*. It returns the empty string "" on end of file. Note that the newline character is left in the returned string.

Math

exp(*d* : double) : double

returns e to the *d*th power.

log(*d* : double) : double

returns the natural log of *d*.

sqrt(*d* : double) : double

returns the square root of the double *d*.

pow(*d* : double, *x* : double) : double

returns *d* raised to the *x*th power.

cos(*d* : double) : double

returns the cosine of *d*.

sin(*d* : double) : double

returns the sine of *d*.

atan2(*y* : double, *x* : double) : double

returns the arctangent of *y/x* in the range -pi to pi.

Miscellaneous

exit() : void

exit(*v* : int) : void

causes **gvpr** to exit with the exit code *v*. *v* defaults to 0 if omitted.

rand() : double

returns a pseudo-random double between 0 and 1.

srand() : int

srand(*v* : int) : int

sets a seed for the random number generator. The optional argument gives the seed; if it is omitted, the current time is used. The previous seed value is returned. **srand** should be called before any calls to **rand**.

BUILT-IN VARIABLES

gvpr provides certain special, built-in variables, whose values are set automatically by **gvpr** depending on the context. Except as noted, the user cannot modify their values.

\$: obj_t

denotes the current object (node, edge, graph) depending on the context. It is not available in **BEGIN** or **END** clauses.

\$F : string

is the name of the current input file.

\$G : graph_t

denotes the current graph being processed. It is not available in **BEGIN** or **END** clauses.

\$O : graph_t

denotes the output graph. Before graph traversal, it is initialized to the target graph. After traversal and any **END_G** actions, if it refers to a non-empty graph, that graph is printed onto the output stream. It is only valid in **N**, **E** and **END_G** clauses. The output graph may be set by the user.

\$T : graph_t

denotes the current target graph. It is a subgraph of **\$G** and is available only in **N**, **E** and **END_G** clauses.

\$tgtname : string

denotes the name of the target graph. By default, it is set to "gvpr_result". If used multiple times during the execution of **gvpr**, the name will be appended with an integer. This variable may be set by the user.

\$troot : node_t

indicates the starting node for a (directed or undirected) depth-first traversal of the graph (cf. **\$tvtype** below). The default value is **NULL** for each input graph.

\$tvtype : tvtype_t

indicates how **gvpr** traverses a graph. At present, it can only take one of six values: **TV_flat**, **TV_dfs**, **TV_fwd**, **TV_ref**, **TV_ne**, and **TV_en**. **TV_flat** is the default. The meaning of these values is discussed below.

ARGC : int

denotes the number of arguments specified by the **-a args** command-line argument.

ARGV : string array

denotes the array of arguments specified by the **-a args** command-line argument. The *i*th argument is given by **ARGV[i]**.

BUILT-IN CONSTANTS

There are several symbolic constants defined by **gvpr**.

NULL : obj_t

a null object reference, equivalent to 0.

TV_flat : tvtype_t

a simple, flat traversal, with graph objects visited in seemingly arbitrary order.

TV_ne : tvtype_t

a traversal which first visits all of the nodes, then all of the edges.

TV_en : tvtype_t

a traversal which first visits all of the edges, then all of the nodes.

TV_dfs : tvtype_t

a traversal of the graph using a depth-first search on the underlying undirected graph. To do the traversal, **gvpr** will check the value of **\$troot**. If this has the same value that it had previously (at the start, the previous value is initialized to **NULL**), **gvpr** will simply look for some unvisited node and traverse its connected component. On the other hand, if **\$troot** has changed, its connected component will be toured, assuming it has not been previously visited or, if **\$troot** is **NULL**, the traversal will stop. Note that using **TV_dfs** and **\$troot**, it is possible to create an infinite loop.

TV_fwd : *tvtype_t*

a traversal of the graph using a depth-first search on the graph following only forward arcs. In *libagraph(3)*, edges in undirected graphs are given an arbitrary direction, which is used for this traversal. The choice of roots for the traversal is the same as described for **TV_dfs** above.

TV_rev : *tvtype_t*

a traversal of the graph using a depth-first search on the graph following only reverse arcs. In *libagraph(3)*, edges in undirected graphs are given an arbitrary direction, which is used for this traversal. The choice of roots for the traversal is the same as described for **TV_dfs** above.

EXAMPLES

```
gvpr -i 'N[color=="blue"]' file.dot Generate the node-induced subgraph of all nodes with color blue. gvpr
-c 'N[color=="blue"]{color = "red"}' file.dot Make all blue nodes red. BEGIN { int n, e; int tot_n = 0; int
tot_e = 0; } BEG_G {
  n = nNodes($G);
  e = nEdges($G);
  printf ("%d nodes %d edges %s0, n, e, $G.name);
  tot_n += n;
  tot_e += e; } END { printf ("%d nodes %d edges total0, tot_n, tot_e) } Version of the program gc. gvpr -c
"" Equivalent to nop. BEG_G { graph_t g = graph ("merge", "S"); } E {
  node_t h = clone(g,$.head);
  node_t t = clone(g,$.tail);
  edge_t e = edge(t,h,"");
  e.weight = e.weight + 1; } END_G { $O = g; } Produces a strict version of the input graph, where the
weight attribute of an edge indicates how many edges from the input graph the edge represents. BEGIN
{ node_t n; int deg[] } E{deg[head]++; deg[tail]++; } END_G {
  for (deg[n]) {
    printf ("deg[%s] = %d0, n.name, deg[n]);
  } } Computes the degrees of nodes with edges.
```

ENVIRONMENT

GPRPATH

Colon-separated list of directories to be searched to find the file specified by the -f option.

BUGS

When the program is given as a command line argument, the usual shell interpretation takes place, which may affect some of the special names in **gvpr**. To avoid this, it is best to wrap the program in single quotes.

The constants **TV_fht**, **TV_dfs**, **TV_fwd**, and **TV_rev**

There is a single global scope, except for formal function parameters, and even these can interfere with the type system. Also, the extent of all variables is the entire life of the program. It might be preferable for scope to reflect the natural nesting of the clauses, or for the program to at least reset locally declared variables. For now, it is advisable to use distinct names for all variables.

If a function ends with a complex statement, such as an IF statement, with each branch doing a return, type checking may fail. Functions should use a return at the end.

The *expr* library does not support string values of (char*)0. This means we can't distinguish between "" and (char*)0 edge keys. For the purposes of looking up and creating edges, we translate "" to be (char*)0, since this latter value is necessary in order to look up any edge with a matching head and tail.

The language inherits the usual C problems such as dangling references and the confusion between '=' and '=='.

AUTHOR

Emden R. Gansner <erg@research.att.com>

SEE ALSO

awk(1), gc(1), dot(1), nop(1), libexpr(3), libagraph(3)