NAME

gvpr − graph pattern scanning and processing language

SYNOPSIS

gvpr [**−icnqV?**] [**−o** *outfile*] [**−a** *args*] [*'prog'* | **−f** *progfile*] [*files*]

DESCRIPTION

gvpr (previously known as **gpr**) 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 *libcgraph*(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]**. Whitespace characters within single or double quoted substrings, or preceded by a backslash, are ignored as separators. In general, a backslash character turns off any special meaning of the following character. Note that the tokens derived from multiple **−a** flags are concatenated.

−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 **GVPRPATH** to look for the file. If **−f** is not given, **gvpr** will use the first non-option argument as the program.

−q Turns off warning messages.

−n Turns off graph read-ahead. By default, the variable **$NG** is set to the next graph to be processed. This requires a read of the next graph before processing the current graph, which may block if the next graph is only generated in response to some action pertaining to the processing of the current graph.

−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**, **END_G** and **END** clauses. There can be any number of **BEG_G**, **N** and **E** statements, the first applied to graphs, the second to nodes, the third to edges. These are separated into blocks, a block consisting of an optional **BEG_G** statement and all **N** and **E** statements up to the next **BEG_G** statement, if any. The top-level semantics of a **gvpr** program are:

```
Evaluate the BEGIN clause, if any.
For each input graph G {
   For each block {
      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.

The blocks are evaluated in the order in which they occur. Within a block, the **N** clauses (**E** clauses, respectively) are evaluated in the order in which the occur. Note, though, that within a block, **N** or **E** clauses may be interlaced, depending on the traversal order.

Predicates and actions are sequences of statements in the C dialect supported by the *expr*(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 **'...'**. **gvpr** accepts C++ comments as well as cpp-type comments. For the latter, if a line begins with a '#' character, the rest of the line is ignored.

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 **[** *type0* **]**

where *type0* is optional. If it is supplied, the parser will enforce that all array subscripts have the specified type. If it is not supplied, objects of all types can be used as subscripts. 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*
forr(*array* **[** *var* **])** *statement*
while(*expression* **)** *statement*
switch(*expression* **)** *case statements*
break [*expression* **]**
continue [*expression* **]**
return [*expression* **]**

Items in brackets are optional.

In the second form of the **for** statement and the **forr** statement, the variable *var* is set to each value used as an index in the specified array and then the associated *statement* is evaluated. For numeric and string indices, the indices are returned in increasing (decreasing) numeric or lexicographic order for **for** (**forr**, respectively). This can be used for sorting.

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 for the purpose of regular expression matching. Patterns use *ksh*(1) file match pattern syntax. (For simple string equality, use the **strcmp** function.

gvpr will attempt to use an expression as a string or numeric value as appropriate. Both C-like casts and function templates will cause conversions to be performed, if possible.

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 *libcgraph*(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.

X : **douible**
the X coordinate of a node. (Assumes the node has a *pos* attribute.)

Y : **douible**
the Y coordinate of a node. (Assumes the node has a *pos* attribute.)

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* in the root graph, or **NULL**.

nxtnode_sg(*sg* : **graph_t**, *n* : **node_t**) : **node_t**
returns the next node after *n* in *sg*, or **NULL**.

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

isSubnode(*sg* : **graph_t**, *n* : **node_t**) : **int**
returns non-zero if node *n* is in (sub)graph *sg*, or zero otherwise.

indegreeOf(*sg* : **graph_t**, *n* : **node_t**) : **int**
returns the indegree of node *n* in (sub)graph *sg*.

outdegreeOf(*sg* : **graph_t**, *n* : **node_t**) : **int**
returns the outdegree of node *n* in (sub)graph *sg*.

degreeOf(*sg* : **graph_t**, *n* : **node_t**) : **int**
returns the degree of node *n* in (sub)graph *sg*.

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* in the root graph. If the graph is undirected, the distinction between head and tail nodes is unimportant. If such an edge already exists, it is returned.

edge_sg(*sg* : **graph_t**, *t* : **node_t**, *h* : **node_t**, *s* : **string**) : **edge_t**
creates an edge with tail node *t*, head node *h* and name *s* in (sub)graph *sg* (and all parent graphs). 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.

isEdge_sg(*sg* : **graph_t**, *t* : **node_t**, *h* : **node_t**, *s* : **string**) : **edge_t**
looks for an edge with tail node *t*, head node *h* and name *s* in (sub)graph *sg*. 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.

isSubedge(*g* : **graph_t**, *e* : **edge_t**) : **int**
returns non-zero if edge *e* is in (sub)graph *sg*, or zero otherwise.

fstout(*n* : **node_t**) : **edge_t**
returns the first outedge of node *n* in the root graph.

fstout_sg(*sg* : **graph_t**, *n* : **node_t**) : **edge_t**
returns the first outedge of node *n* in (sub)graph *sg*.

nxtout(*e* : **edge_t**) : **edge_t**
returns the next outedge after *e* in the root graph.

nxtout_sg(*sg* : **graph_t**, *e* : **edge_t**) : **edge_t**
returns the next outedge after *e* in graph *sg*.

fstin(*n* : **node_t**) : **edge_t**
returns the first inedge of node *n* in the root graph.

fstin_sg(*sg* : **graph_t**, *n* : **node_t**) : **edge_t**
returns the first inedge of node *n* in graph *sg*.

nxtin(*e* : **edge_t**) : **edge_t**
returns the next inedge after *e* in the root graph.

nxtin_sg(*sg* : **graph_t**, *e* : **edge_t**) : **edge_t**
returns the next inedge after *e* in graph *sg*.

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

fstedge_sg(*sg* : **graph_t**, *n* : **node_t**) : **edge_t**
returns the first edge of node *n* in graph *sg*.

nxtedge(*e* : **edge_t**, **node_t**) : **edge_t**
returns the next edge after *e* in the root graph.

nxtedge_sg(*sg* : **graph_t**, *e* : **edge_t**, **node_t**) : **edge_t**
returns the next edge after *e* in the graph *sg*.

opp(*e* : **edge_t**, **node_t**) : **node_t**
returns the node on the edge *e* not equal to *n*. Returns NULL if *n* is not a node of *e*. This can be useful when using **fstedge** and **nxtedge** to enumerate the neighbors of *n*.

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*.

cloneG(*g* : **graph_t**, *s* : **string**) : **graph_t**
creates a clone of graph *g* with name of *s*. If *s* is "", the created graph has the same name as *g*.

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. In this case, the call is equivalent to **cloneG(***x***,"")**.

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.

hasAttr(*src* : **obj_t**, *name* : **string**) : **int**
: returns non-zero if object *src* has an attribute whose name is *name*. It returns 0 otherwise.

isAttr(*g* : **graph_t**, *kind* : **string**, *name* : **string**) : **int**
: returns non-zero if an attribute *name* has been defined in *g* for objects of the given *kind*. For nodes, edges, and graphs, *kind* should be "N", "E", and "G", respectively. It returns 0 otherwise.

aget(*src* : **obj_t**, *name* : **string**) : **string**
: returns the value of attribute *name* in object *src*. This is useful for those cases when *name* conflicts with one of the keywords such as "head" or "root". If the attribute has not been declared in the graph, the function will initialize it with a default value of "". To avoid this, one should use the **hasAttr** or **isAttr** function to check that the attribute exists.

aset(*src* : **obj_t**, *name* : **string**, *value* : **string**) : **int**
: sets the value of attribute *name* in object *src* to *value*. Returns 0 on success, non-zero on failure. See **aget** above.

getDflt(*g* : **graph_t**, *kind* : **string**, *name* : **string**) : **string**
: returns the default value of attribute *name* in objects in *g* of the given *kind*. For nodes, edges, and graphs, *kind* should be "N", "E", and "G", respectively. If the attribute has not been declared in the graph, the function will initialize it with a default value of "". To avoid this, one should use the **isAttr** function to check that the attribute exists.

setDflt(*g* : **graph_t**, *kind* : **string**, *name* : **string**, *value* : **string**) : **int**
: sets the default value of attribute *name* to *value* in objects in *g* of the given *kind*. For nodes, edges, and graphs, *kind* should be "N", "E", and "G", respectively. Returns 0 on success, non-zero on failure. See **getDflt** above.

fstAttr(*g* : **graph_t**, *kind* : **string**) : **string**
: returns the name of the first attribute of objects in *g* of the given *kind*. For nodes, edges, and graphs, *kind* should be "N", "E", and "G", respectively. If there are no attributes, the string "" is returned.

nxtAttr(*g* : **graph_t**, *kind* : **string**, *name* : **string**) : **string**
: returns the name of the next attribute of objects in *g* of the given *kind* after the attribute *name*. The argument *name* must be the name of an existing attribute; it will typically be the return value of an previous call to **fstAttr** or **nxtAttr**. For nodes, edges, and graphs, *kind* should be "N", "E", and "G", respectively. If there are no attributes left, the string "" is returned.

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*.

kindOf(*obj* : **obj_t**) : **string**
: returns an indication of the type of *obj*. For nodes, edges, and graphs, it returns "N", "E", and "G", respectively.

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.

strcmp(*s1* : **string**, *s2* : **string**) : **int**
provides the standard C function *strcmp*(3).

length(*s* : **string**) : **int**
returns the length of string *s*.

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

rindex(*s* : **string**, *t* : **string**) : **int**
returns the index of the character in string *s* where the leftmost (rightmost) 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*.

toupper(*s* : **string**) : **string**
returns a version of *s* with the alphabetic characters converted to upper-case.

tolower(*s* : **string**) : **string**
returns a version of *s* with the alphabetic characters converted to lower-case.

canon(*s* : **string**) : **string**
returns a version of *s* appropriate to be used as an identifier in a dot file.

html(*g* : **graph_t**, *s* : **string**) : **string**
returns a ‘‘magic’’ version of *s* as an HTML string. This will typically be used to attach an HTML-like label to a graph object. Note that the returned string lives in *g*. In particular, it will be freed when *g* is closed, and to act as an HTML string, it has to be used with an object of *g*. In addition, note that the angle bracket quotes should not be part of *s*. These will be added if *g* is written in concrete DOT format.

ishtml(*s* : **string**) : **int**
returns non-zero if and only if *s* is an HTML string.

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.

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

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

sscanf(*s* : **string**, *fmt* : **string**, ...) : **int**
scans the string *s*, extracting values according to the *sscanf*(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.

split(*s* : **string**, *arr* : **array**, *seps* : **string**) : **int**

split(*s* : **string**, *arr* : **array**) : **int**

tokens(*s* : **string**, *arr* : **array**, *seps* : **string**) : **int**

tokens(*s* : **string**, *arr* : **array**) : **int**
The **split** function breaks the string *s* into fields, while the **tokens** function breaks the string into tokens. A field consists of all non-separator characters between two separator characters or the beginning or end of the string. Thus, a field may be the empty string. A token is a maximal, non-empty substring not containing a separator character. The separator characters are those given in the *seps* argument. If *seps* is not provided, the default value is " \t\n". The functions return the number of fields or tokens.

The fields and tokens are stored in the argument array. The array must be **string**-valued and have **int** as its index type. The entries are indexed by consecutive integers, starting at 0. Any values already stored in the array will be either overwritten, or still be present after the function returns.

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 *fopen*(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.

MIN(*y* : **double**, *x* : **double**) : **double**
: returns the minimum of *y* and *x*.

MAX(*y* : **double**, *x* : **double**) : **double**
: returns the maximum of *y* and *x*.

Associative Arrays

*arr* : **int**
: returns the number of elements in the array *arr*.

idx **in** *arr* : **int**
: returns 1 if a value has been set for index *idx* in the array *arr*. It returns 0 otherwise.

unset(*v* : **array**, *idx*) : **int**
: removes the item indexed by *idx*. It returns 1 if the item existed, 0 otherwise.

unset(*v* : **array**) : **void**
: re-initializes the array.

Miscellaneous

exit(*v* : **int**) : **void**
: causes **gvpr** to exit with the exit code *v*.

system(*cmd* : **string**) : **int**
: provides the standard C function *system*(3). It executes *cmd* in the user's shell environment, and returns the exit status of the shell.

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**.

colorx(*color* : **string**, *fmt* : **string**) : **string**
translates a color from one format to another. The *color* argument should be a color in one of the recognized string representations. The *fmt* value should be one of "RGB", "RGBA", "HSV", or "HSVA". An empty string is returned on error.

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.

$NG : **graph_t**
denotes the next graph to be processed. If **$NG** is NULL, the current graph **$G** is the last graph. Note that if the input comes from stdin, the last graph cannot be determined until the input pipe is closed. It is not available in **BEGIN** or **END** clauses, or if the **-n** flag is used.

$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.

$tvroot : **node_t**
indicates the starting node for a (directed or undirected) depth-first or breadth-first traversal of the graph (cf. **$tvtype** below). The default value is **NULL** for each input graph. After the traversal at the given root, if the value of **$tvroot** has changed, a new traversal will begin with the new value of **$tvroot**. Also, set **$tvnext** below.

$tvnext : **node_t**
indicates the next starting node for a (directed or undirected) depth-first or breadth-first traversal of the graph (cf. **$tvtype** below). If a traversal finishes and the **$tvroot** has not been reset but the **$tvnext** has been set but not used, this node will be used as the next choice for **$tvroot**. The default value is **NULL** for each input graph.

$tvedge : **edge_t**
For BFS and DFS traversals, this is set to the edge used to arrive at the current node or edge. At the beginning of a traversal, or for other traversal types, the value is **NULL**.

$tvtype : **tvtype_t**
indicates how **gvpr** traverses a graph. It can only take one of the constant values with the previx "TV_" described below. **TV_flat** is the default.

In the underlying graph library *cgraph*(3), edges in undirected graphs are given an arbitrary direction. This is used for traversals, such as **TV_fwd**, requiring directed edges.

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*
TV_postdfs : *tvtype_t*
TV_prepostdfs : *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 **$tvroot**. If this has the same value that it had previously (at the start, the previous value is initialized to **NULL**.), **gvpr** will simply look for some unvisited node and traverse its connected component. On the other hand, if **$tvroot** has changed, its connected component will be toured, assuming it has not been previously visited or, if **$tvroot** is **NULL**, the traversal will stop. Note that using **TV_dfs** and **$tvroot**, it is possible to create an infinite loop.

By default, the traversal is done in pre-order. That is, a node is visited before all of its unvisited edges. For **TV_postdfs**, all of a node's unvisited edges are visited before the node. For **TV_prepostdfs**, a node is visited twice, before and after all of its unvisited edges.

TV_fwd : *tvtype_t*
TV_postfwd : *tvtype_t*
TV_prepostfwd : *tvtype_t*
A traversal of the graph using a depth-first search on the graph following only forward arcs. The choice of roots for the traversal is the same as described for **TV_dfs** above. The different order of visitation specified by **TV_fwd**, **TV_postfwd** and **TV_prepostfwd** are the same as those specified by the analogous traversals **TV_dfs**, **TV_postdfs** and **TV_prepostdfs**.

TV_rev : *tvtype_t*
TV_postrev : *tvtype_t*
TV_prepostrev : *tvtype_t*
A traversal of the graph using a depth-first search on the graph following only reverse arcs. The choice of roots for the traversal is the same as described for **TV_dfs** above. The different order of visitation specified by **TV_rev**, **TV_postrev** and **TV_prepostrev** are the same as those specified by the analogous traversals **TV_dfs**, **TV_postdfs** and **TV_prepostdfs**.

TV_bfs : *tvtype_t*
A traversal of the graph using a breadth-first search on the graph ignoring edge directions. See the item on **TV_dfs** above for the role of **$tvroot**.

EXAMPLES

```
gvpr −i 'N[color=="blue"]' file.gv
```

Generate the node-induced subgraph of all nodes with color blue.

```
gvpr −c 'N[color=="blue"]{color = "red"}' file.gv
```

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 %s\n", n, e, $G.name);
  tot_n += n;
  tot_e += e;
}
END { printf ("%d nodes %d edges total\n", 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] = %d\n", n.name, deg[n]);
  }
}
```

Computes the degrees of nodes with edges.

```
BEGIN {
  int i, indent;
  int seen[string];
  void prInd (int cnt) {
    for (i = 0; i < cnt; i++) printf ("  ");
  }
}
BEG_G {

   $tvtype = TV_prepostfwd;
   $tvroot = node($,ARGV[0]);
```

```
}
N {
  if (seen[$.name]) indent--;
  else {
    prInd(indent);
      print ($.name);
    seen[$.name] = 1;
    indent++;
  }
}
```

Prints the depth-first traversal of the graph, starting with the node whose name is **ARGV[0]**, as an indented list.

ENVIRONMENT

GVPRPATH

Colon-separated list of directories to be searched to find the file specified by the −f option. **gvpr** has a default list built in. If **GVPRPATH** is not defined, the default list is used. If **GVPRPATH** starts with colon, the list is formed by appending **GVPRPATH** to the default list. If **GVPRPATH** ends with colon, the list is formed by appending the default list to **GVPRPATH**. Otherwise, **GVPRPATH** is used for the list.

On Windows systems, replace ‘‘colon’’ with ‘‘semicolon’’ in the previous paragraph.

BUGS AND WARNINGS

Scripts should be careful deleting nodes during **N{}** and **E{}** blocks using BFS and DFS traversals as these rely on stacks and queues of nodes.

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.

If string constants contain pattern metacharacters that you want to escape to avoid pattern matching, two backslashes will probably be necessary, as a single backslash will be lost when the string is originally scanned. Usually, it is simpler to use **strcmp** to avoid pattern matching.

As of 24 April 2008, **gvpr** switched to using a new, underlying graph library, which uses the simpler model that there is only one copy of a node, not one copy for each subgraph logically containing it. This means that iterators such as *nxtnode* cannot traverse a subgraph using just a node argument. For this reason, subgraph traversal requires new functions ending in "_sg", which also take a subgraph argument. The versions without that suffix will always traverse the root graph.

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.

Related to this, strings converted to integers act like char pointers, getting the value 0 or 1 depending on whether the string consists solely of zeroes or not. Thus, the ((int)"2") evaluates to 1.

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), expr(3), cgraph(3)