The DGL Manual

Peter M. Maurer

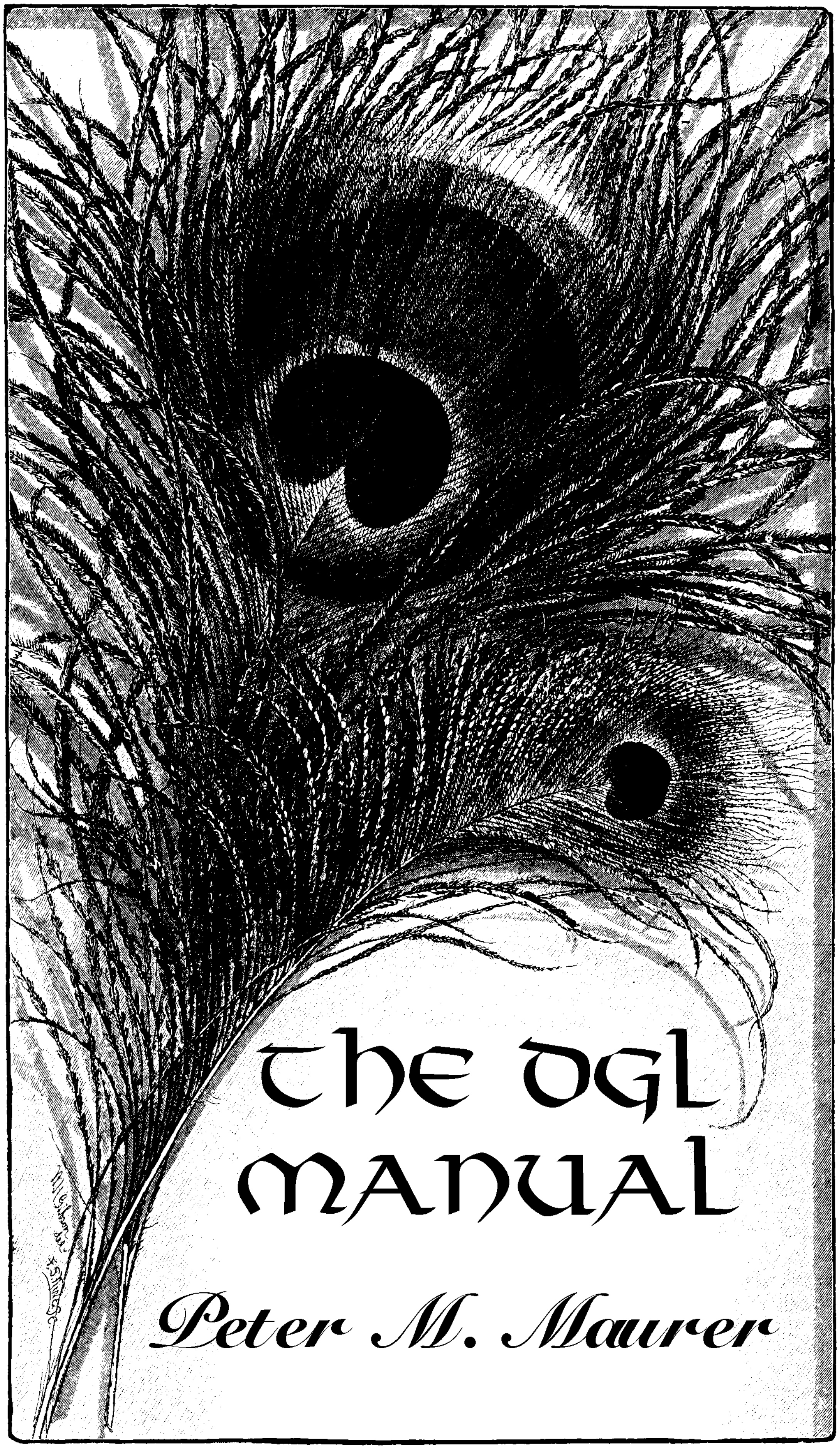

TABLE OF CONTENTS

REFERENCE MANUAL FOR A DATA GENERATION LANGUAGE BASED ON PROBABILISTIC CONTEXT FREE GRAMMARS

Version 1.0

Peter M. Maurer
Department of Computer Science and Engineering
University of South Florida
Tampa, FL 33620

1. Introduction.

This report describes the data generation language "dgl." The language was originally designed to general functional level tests for VLSI designs, but there is no inherent reason why this tool cannot be used for other purposes. In particular, there is no reason why this tool could not be used to generate tests for software systems as well as for other types of hardware. The tool is particularly adapted to situations requiring random selection and printing of data. Some frivolous uses that the to has been put to are dealing bridge hands, and printing daily fortune messages at login time. Despite this, the tool is intended to be a partial solution to one of the most difficult problems encountered in a VLSI design, namely that of verifying the correctness of the design at the highest level.

The dgl language was design to facilitate the construction of data generators that select items at random from a set of items described by a probabilistic context free grammar. Since many tests contain context sensitive data, or data that is difficult to describe using a context free grammar, dgl also provides several features for generating non-context free languages. Once the format of the test data has been described in dgl, the dgl compiler can be used to create a data-generator based on the grammar. This data-generator can then be used to saturate a VLSI design with bias-free tests.

2. Productions.

The basic descriptive unit of dgl is the production. Productions are used to describe the data that is to be output, and the data generator produces data items by interpreting productions. The simplest productions have the following form.

```
<name>: <string>, ... ,<string>;
```

The element <name>, which is the name of the production, must be a string of letters, digits and underlines, and should start with a letter. The rules for forming the <string> elements are given in the next section. The data generator operates by selecting one alternative from the production named "main" and interpreting it. Each <string>

constitutes one alternative. In the simplest case, the string is simply output as in the following example.

main: abc,def,ghi,jkl;

In this case, the data generator will simply output one of the four alternative strings and stop. The situation becomes more complicated when the strings contain references to other productions. These references are known as non-terminals and are of the form %x or %{name} where x is a single character and "name" is an arbitrarily long string. When a non-terminal is found, the interpretation of the current string is suspended, and an alternative is chosen from the referenced production. This alternative is completely interpreted before the interpretation of the first string is resumed. It is possible to nest references arbitrarily deep.

By default alternatives are chosen with equal probability, but it is possible to weight the alternatives so some of them will be chosen more often than others. The following is an example of a production with weighted alternatives.

main: 1: abc, 2:def, 3:ghi, 4:jkl;

When the data generator chooses an alternative from this production, "def" is twice as likely to be chosen as"abc" while "ghi" is three times as likely. If one alternative is weighted, then all must be weighted. Since the underlying implementation of of unweighted productions is simpler than that of weighted productions, the use of weighted productions with all-equal weights should be avoided.

Finally, a production may have an arbitrary number of alternatives, but no two productions can have the same name.

3. The Rules for Forming Strings.

Dgl has a number of features that make it easy to form sets of alternatives. The most straightforward way is to write all alternatives out separated by commas. Strings may be enclosed in single or double quotes (an apostrophe is a single quote), but the quotes may be omitted if the strings contain no illegal characters. The illegal characters are colons, commas, semicolons parentheses, square brackets, dashes, exclamation points, double and single quotes, spaces, tabs, and newlines. Any other printable character may appear in a quoted string. If the backslash character is the first character of a string, it will be deleted and any special meaning attached to the string will be ignored. This feature allows dgl keywords to be used as alternatives in a production by preceding them with a backslash as in \macro. HOWEVER, strings containing dgl reserved *characters*, such as the semi-colon (;) *must be enclosed in quotes,* or the special meaning of the characters will not be removed. Keywords may also be used in a string if they are enclosed in quotes. If it is necessary to begin an unquoted string with a backslash, two consecutive backslashes must be used.

Strings protected by quotes may contain any character as long as certain conventions are observed. If a string contains a double quote, the double quote must be preceded by a backslash. It is also a good idea to precede single quotes by backslashes, but this is not necessary unless the string is protected by single quotes. The backslash will not appear in the output. A quoted string must begin and end on a single line. If it is necessary to include newline characters in a string, the sequence \n should be used. The rules for backslashes are the same as those for C programs, with the exception that the sequence \0 must not be used in a string. To summarize, \n represents a newline, \t represents a tab, \r a carriage return, \b a backspace, \f a form feed, \\ a backslash, and \ddd (where ddd is three octal digits) an arbitrary 8-bit character other than \000. In all other cases the sequence \c where c is an arbitrary character, represents the character c.

If a string is too long to fit on one line, it may be broken at any convenient place, and the separate pieces placed on consecutive lines. If the string is quoted, place a quote at the end of the first line, and begin the continuation with a new quote. No comma is placed between the strings. When the dgl compiler encounters two strings that are not separated by commas, it simply concatenates them (beware of missing commas). Therefore abc, a b c, and "a" "bc" all mean the same thing. There is a hidden limit of 500 characters on the length of strings, but if strings are formed by concatenation, the limit rises to 5000 characters. There is no limit on the length of strings formed at run time, so if strings longer than 5000 characters are needed, they can be formed by using non-terminals.

Dgl also provides several convenient methods for generating a set of alternatives with a single string. For example, the set of alternatives a,b,c,d,e,f,g can also be written [a-g]. The following production can be used to select letters of the alphabet, one at a time.

```
letter: [a-zA-Z];
```

The construct [<string>] (a string enclosed in square brackets) is called a character set. Each item in a character set represents one single-character alternative. The simplest form of the character set contains no dashes as in [abxyz] which is the same as a,b,x,y,z. When the character set contains a dash, a range of characters is generated. This range includes the character preceding the dash, the character following the dash, and all characters in between. When it makes a difference, characters are generated consecutively either high-to-low or low-to-high, depending on the order of the beginning and ending characters. Thus [a-f] is the same as a,b,c,d,e,f and [f-a] is the same as f,e,d,c,b,a. It is possible for a character set to contain duplicates, so [aaa] is the same as a,a,a. Spaces tabs and newlines in a character set are ignored, so [a b c] is the same as [abc]. When a character set is concatenated with an ordinary string, the string is replicated for each member of the character set. Thus part[abc] is the same as parta, partb, partc. It is possible to concatenate more than one character set in a string as in [a-z]=[a-z] which is the same as a=a,a=b, ... z=y,z=z. If it is necessary to include any of the illegal characters in a string, they must be enclosed in quotes, and the quotes must appear inside the square brackets, as in ["[]()-"]. It is not a good idea to try to use the dash along with illegal characters, but if you must, something of the form ["!(),:;a"-z] will work. In a character set, a backslash usually represents itself. One cannot include things like newlines and returns in character sets. One exception is a backslash that is preceded by a tab, space, newline, "[" character, a dash, or the closing quote of the preceding string. These backslashes disappear from the character set. If a character set looks like a dgl keyword, one of the conventions mentioned above must be used to remove the special meaning. Thus [macro] is illegal, but [\macro], ["macro"] and [m a c r o] are all legal and all mean the same thing.

The macro is an extension of the concept of the character set which may be used to define sets of strings as well as sets of characters. Unlike the character set, the macro is always declared separately from the production in which it is used. The rules for defining macros are identical to those for defining unweighted productions, except for the "macro" keyword which must follow the name of the production as illustrated below.

```
abcs: macro abc[123],abc100;
```

The definition of a macro must always precede its use. Once it has been defined, a macro may be used in any place where a character set is permitted. One uses a macro by preceding the name with an exclamation point. The name must be followed by a space or some other illegal character to indicate where the name ends. The following is an example of a production that uses the macro defined above.

```
useabcs: head_!abcs;
```

This production could also have been coded as head_abc1,head_abc2, ... The rules for concatenating macros are the same as those for concatenating character sets. The current implementation of dgl allows any ordinary unweighted production to be used as a macro, as long as the definition of the production precedes its use as a macro.

In the above specification, the number of characters in a string is determined by the length of the specification. However, if it is necessary to specify fixed length strings, a width specification can be inserted ahead of the first string of any alternative. When concatenation is used to form an alternative, the width specification applies to the length of the *alternative* not to the length of the strings that make up the alternative. When a width specification is used, the width applies to all alternatives following the width specification up to either the end of the production or to the next alternative. A width specification of zero is used to turn off the fixed width option. An alternative will be padded on the right with spaces, or truncated on the right to adjust it to the specified width. The effect of a fixed width specification always terminates at the end of a production. An example of a width specification is given below.

```
a: "abc", "g" , width(4) , "uvwxyz", a, m a c r o;
```

This is equivalent to specifying the following.

```
a: "abc", "g", "uvwx", "a   ", "macr";
```

4. More Types of Productions.

Dgl provides shorthand methods for common types of constructions. For example suppose it is necessary to create numbers that contain mostly zeros. The following two productions can be used to accomplish this.

```
numbers: 3:0, 1:%{nonzero};
nonzero: [1-9];
```

Dgl allows this sort of thing to be done with a single production as follows.

```
numbers: 3:0, 1:(1,2,3,4,5,6,7,8,9);
```

When a list of strings is enclosed in parentheses following a weight, the weight applies to the list, not to the individual elements. Items within the list are selected with equal probability. If the set of strings can be completely specified without using commas, then the enclosing parentheses may be omitted as illustrated below.

```
numbers: 3:0, 1:[1-9];
```

Another common requirement is to select numbers from a given range with equal probability. This is extremely cumbersome with ordinary productions. Dgl provides the "range" construct for simplifying this type of specification. The following production causes numbers to be selected from the range 1 through 100, inclusive.

```
example: range 1,100;
```

In this example, the one and two digit numbers will be printed without leading zeros. If it is desirable to have all numbers the same length, the "width" parameter can be added.

```
example: range width(5) 1,100;
```

This example will generate 5-digit numbers with leading zeros where necessary. If the specified width is too short for the generated number, leading digits will be truncated. The "range" construct allows either number to be negative, and allows a range to be specified as a single number as illustrated below.

```
example: range 100;
```

When a single number is specified, it is assumed that the first number has been omitted, which then defaults to zero. When using negative numbers in a range, it is important to remember that the first number must be less than the first, as in the following example.

```
example: range -100,0
```

If the numbers specifying a range are specified out of order, the range production will produce the single number "0" each time it is interpreted.

5. More on Non-Terminals.

The two basic types of non-terminals are %x and %{name}. These non-terminals may be augmented with a repetition count that causes several selections to be made using the same production. For example, %5x causes five consecutive selections to be made from the production "x." The non-terminal %5x is logically equivalent to the sequence %x%x%x%x%x. When a count is used with the second form of non-terminal, the count appears outside the curly brackets as in %5{name}. In place of a count, a range can be used to make a random number of selections. A range can be specified in two ways, exemplified by %5-7x, and %5.7x. A range consists of two numbers separated by either a period or a dash. The second number must be greater than or equal to the first. For example, the non-terminal %3.5x will cause either 3, 4, or 5 selections to be made from the production "x." The numbers 3, 4, and 5 will be chosen with equal probability. When a dash is used as the separating character of the range, the non-terminal must be enclosed in quotes to remove the special meaning attached to the character "-". Since non-terminals are interpreted at run time, enclosing them in quotes does not remove their special meaning.

Since quoting a non-terminal does not remove its special meaning, it is necessary to use a different convention to include data that looks like non-terminals in the output. If the character % is needed in the output, the sequence %% must be used. The sequence %% appears as a single % in the output. This is a special case of non-terminals that refer to undefined productions. If the data generator encounters the sequence "%x" and there is n o production named "x" then the generator will output the character "x". Similarly, if it encounters %{name} and "name" is not defined, then it will output the string "name". Repetition counts and ranges will be applied to these strings, so %5x will cause xxxxx to be output, while %2{name} will cause namename to be output. When a non-terminal is used in this manner, it is permissible for it to contain special characters. It cannot, of course, contain curly braces.

Curly braces are significant only after the % character. If they appear anywhere else in a string they will be treated as output characters. If the data generator encounters the sequence "%{" and there is no matching "}" character, the entire rest of the string will be treated as the production name. If the rest of the string does not contain a name (terminates with “%25.40{” for example), the sequence of characters starting with the % is treated as output characters. If ranges are specified as "123." or ".123" the omitted number will be treated as a zero. A zero repetition count, a zero selected from a range, and a range with ending number smaller than the beginning number will cause the non-terminal to be ignored. If a range contains more than one dash, the second dash and everything following, up to the production name or curly brace, will be ignored. The numbers in a range cannot be negative. If an alternative ends with a sequence of the form "%" or "%{" or "%25" or "%25{" or "%25-200" or "%25-200{" this sequence will be treated as data.

6. Techniques for Systematic Generation of Data.

An ordinary production allows duplicates to be selected. For example, each time the following production is used, the probability of each of the 26 alternatives remains the same.

```
abc: [a-z];
```

For some applications it is necessary to restrict the number of duplicates that can be selected from each production. If it is desirable to select each letter once and only once, but in random order, the following production must be used.

```
abc: unique [a-z];
```

The "unique" keyword will cause a different letter to be selected each time the production is used. Letters are chosen at random from the set of unused letters, with equal probability.

Sometimes it is useful to limit the number of selections of a certain alternative, but not restrict that number to 1. For example, it may be desirable to choose five a's and three b's in random order. One way to do this is as follows.

```
example: unique a,a,a,a,a, b,b,b;
```

Dgl allows the following shorthand to be used.

```
example: unique 5:a, 3:b;
```

When this form is used, every alternative must have a repetition count, even if that count is 1. If several alternatives have the same repetition count, those alternatives may be grouped as in the following example.

```
example: unique 5:(a,b,c), 3:(c,d);
```

Dgl provides several other types of productions for systematically generating data. Suppose it is necessary to select alternatives sequentially instead of randomly.This can be done with the following construct.

```
example: sequence [a-z];
```

The "sequence" keyword causes alternatives to be selected sequentially in the order specified. Both the "unique" construct and the "sequence" construct are identical to ordinary productions, except for the keyword.

The "counter" construct is the sequential counterpart of the "range" construct. The following is an example of a counter that produces the sequence 1,3,5,7,9.

```
ex: counter 1,9,2;
```

The three numbers following the "counter" keyword are the starting number, the ending number and the increment, respectively. Any or all of the three numbers may be omitted. If the first is omitted, it defaults to 1. If the second is omitted, it defaults to "no ending value," and if the third is omitted, it also defaults to 1. Any of the numbers may be negative. If the increment is negative, the second number must be omitted or be smaller than the first number. If the combination of the starting number and increment make it impossible to hit the ending number exactly, the largest number generated will be strictly less (greater for negative increments) than the ending number.

Normally numbers are output without leading zeros, so numbers with fewer digits occupy less space than those with more. If it is necessary for all numbers to be the same length, the "width" keyword should be used as illustrated below.

```
ex: counter width(5) ,100000;
```

If the generated number is shorter than the width, it will be padded on the left with leading zeros. If it is longer than the width, high-order digits will be truncated. For negative numbers, the high-order leading zero will be replaced by a minus sign. If the number of significant digits is equal to or greater than the width, the minus sign will be truncated.

When the start, end, and increment numbers are omitted, it may be necessary to specify leading or consecutive commas to indicate the position of the omitted number. Trailing commas should never be specified. The legal combinations are ",n" ",,n" and "n" for two numbers omitted, and ",n,m" "n,m" and "n,,m" for one number omitted. If all three numbers are omitted, all commas must also be omitted.

At times it will be necessary to enumerate all combinations of certain types of data. For example, suppose it is necessary to generate a list of names. We wish to use the first names "John," "Mary," "Fred," and "Rose." We wish to use an arbitrary letter for the middle initial, and "Smith," "Jones," and "Brown" for last names. We want to generate full names for all combinations of these first names, last names, and middle initials. The following set of productions allows us to do this.

```
full: chain %{first} " " %{mi} " " %{last};
first: chain John,Mary,Fred,Rose;
mi: chain [A-Z].;
last: chain Smith,Jones,Brown;
```

These productions are identical to ordinary productions except for the "chain" keyword. In its simplest form, the "chain" construct is identical to the "sequence" construct. For example, if the "first" production of the above example were referenced by an ordinary production, it would produce the sequence "John", "Mary", "Fred", "Rose". However, when a "chain" production is referenced by another "chain," the sequential selection of alternatives is coordinated with the selection of alternatives from the referencing production as well as with the selection of of alternatives from other "chain" productions referenced either directly or indirectly by the first chain production. The easiest way to thing about the "chain" construction is to thing of the highest level production as defining a set of strings. The data generator produces these strings in a sequential manner. When successive strings are generate, the rightmost production at the greatest depth varies the fastest. In most cases the highest level production will define only a finite number of strings. However, it is also possible to use the "chain" construct to define infinite languages. There is no restriction on how such a language may be defined, but to be able to enumerate all elements of such a set, special precautions must be taken in how the "chain" productions are constructed. If the highest-level production has more than one alternative, only the last alternative should define an infinite number of strings. If the last alternative has more than one non-terminal, only the first non-terminal should define an infinite number of substrings. These rules apply recursively to any "chain" production referenced by the highest level production, directly or indirectly. The following is an example of a specification that enumerates strings of the form a...ab...b with an equal number of a's and b's. The strings are enumerated from shortest to longest.

```
s: chain "", a%sb;
```

The easiest way to use the "chain" construct is to start with an ordinary grammar defining the data that is to be generated, and add the "chain" keyword to those productions that should have every possibility enumerated. Then add the "chain" keyword to the highest level production to coordinate the selections from the lower-level productions.

The state of all productions in a "chain" construct is associated *only* with the highest level production. Furthermore, the association between related "chain" productions is constructed at run time. This implies that the state of a "chain" production is different depending on the highest level of production used to reference it. To illustrate, consider the following example.

```
m: %s%s%t%t%u%u;
s: chain x%u,y%u,z%u;
t: chain q%u,e%u,d%u;
u: chain a,b,c;
```

The first reference to "m" will produce the string xaxbqaqbab. Note that the production u has three internal states, one associated with indirect references through "s", one with indirect references through "t", and one associated with direct references. When it makes a difference, the state of all indirectly referenced productions is considered to be part of the state of the highest level production.

If an indirectly referenced production is referenced more than once by the highest level production, there is a separate state maintained for each reference. Thus the following example generates all combinations of the characters "a", "b" and "c".

```
letters: chain %s%s%s;
s: chain a,b,c;
```

This rule also applies if the references are on different levels. Each distinct reference to a "chain" production in the derivation tree of a string is assigned a distinct state. One caution, a "chain" production should not reference another chain production using a non-terminal of the form %5-7s (%5s is ok). If this type of non-terminal is used, unpredictable results will occur. The same effect can be accomplished by using an intermediate "chain" production, which will work correctly.

Although chains are quite useful for generating many types of systematic data, only one chain can be active at a time. This can be a drawback when some component of systematically generated data must itself be systematically generated, but not coordinated with selections from the primary set of chain productions. Suppose for example, it is necessary to number a set of tests in ascending order using 5-digit base-3 numbers.

(Admittedly, this example is somewhat artificial, but most useful examples are far to complex for an introductory document such as this.)

The following set of productions can be used to generate the base-3 numbers.

```
base3: chain %5{b3digit};
b3digit: chain 0,1,2;
```

Now suppose the tests themselves consist of 8-character strings containing the letters "a" and "b," or 8-character strings containing the letters "c" and "d." These tests can be systematically generated using the following set of productions.

```
main: chain %{abcd};
abcd: chain %8{ab},%8{cd};
ab: chain a,b;
cd: chain c,d;
```

Now suppose the data must be output in the following form.

```
00000. aaaaaaaa
00001. aaaaaaab
00002. aaaaaaba
00010. aaaaaabb
(etc.)
```

Now suppose one attempts to combine the two above sets of productions into a single grammar in the following manner, with the hope of generating the desired data.

```
main: chain %{base3} ". " %{abcd} "\n";
abcd: chain %8{ab},%8{cd};
ab: chain a,b;
cd: chain c,d;
base3: chain %5{b3digit};
b3digit: chain 0,1,2;
```

This grammar actually generates the following data.

```
00000. aaaaaaaa
00000. aaaaaaab
00000. aaaaaaba
00000. aaaaaabb
(etc.)
```

A new line number will not be generated until all choices from "abcd" have been exhausted. The primary problem is that the chain for "base3" is automatically coordinated with the chain starting at "main." The "head" production is designed to allow more than one chain to be active simultaneously. By changing the above grammar as follows, the desired data can be generated.

```
main: chain %{base3} ". " %{abcd} "\n";
abcd: chain %8{ab},%8{cd};
ab: chain a,b;
cd: chain c,d;
base3: head %5{b3digit};
b3digit: chain 0,1,2;
```

When a "head" production is encountered, a new chain is automatically begun. If a chain is already active, the selections from the new chain will not be coordinated with the selections from the existing chain. In all other respects, the "head" production and the "chain" production are identical.

7. Running out of Choices.

All systematic methods for selecting production alternatives suffer from the same problem. At some point the number of choices will be exhausted. There are five selectable actions that can take place when a systematic production runs out of choices. The actions may be different for different productions. The actions are "continue," "restart," "stop," "abort," and "next." These keywords always appear immediately after the keyword that defines the production type. The default for "counter" and "sequence" productions is "restart," while the default for all others is "stop." The "restart" option causes the production to be reset to its initial state when all choices have been exhausted. The "stop" option causes all further selections from the "main" production to be suppressed (see section 9). The "stop" option is intended to be used when it is necessary to generate everything until the set of available choices is exhausted and then stop. The "restart" option is intended to be used when it is necessary to generate a sequence of choices repeatedly.

The "abort" option causes immediate program termination if an attempt is made to reference a production after all choices have been exhausted. The "continue" option causes the referencing non-terminal to be replaced by the null string when no more choices remain. The "next" option allows an alternative production to be named that will be used when no more choices remain in the current production. If this production is not defined, any further reference to the exhausted production will produce a string which is identical to the name of the undefined production. To illustrate the "next" option, consider the following example which causes all lower case letters to be generated in random order, followed by all upper case letters in random order.

```
first: unique next(second) [a-z];
second: unique [A-Z];
```

Any production that can run out of choices can have one of these five options attached to it. If a "counter" production has both an "end-of-choices" and a "width" specification, the "end-of-choices" must come first. In a "chain" construct, an end-of-choices option is ignored unless it is attached to the highest-level production.

The types of productions mentioned so far that can have an end-of-choices option are "unique," "counter," "sequence," "head," and "chain." See section 14 for other types of productions that may have an end-of-choices option.

8. Variables.

Variables can be used to generate non-context-free data, and can be used to generate context-free data that cannot be conveniently generated by any other means. The primary use of variables is when it is necessary to general an item of data at random, and insert that item into the output at several different places. A variable is declared in the following way.

```
x: variable;
```

A value is assigned to a variable by using a non-terminal of the form %{y.x}. This form of non-terminal causes the data generator to select an alternative from the production "y" and assign the choice to "x." This form of non-terminal does not generate output. The alternative chosen from "y" is completely interpreted before being assigned to "x." Thus under normal circumstances, the string assigned to "x" will not contain non-terminals. The value of the variable is accessed by using it like a normal production. Thus the non-terminal %x will cause the current value of the variable "x" to be inserted into the output. The value of a variable can be assigned to another variable, and variable assignments can be nested, in the sense that when the alternative from the production "y" is interpreted, it may have intermediate assignments. Nested variable assignments will usually work correctly even if the nested reference is to the variable currently being assigned.

As stated above, the value of a variable will not normally contain non-terminals. This can be circumvented by assigning a string containing two consecutive % characters, such as "%%x", to the variable. When this string is assigned to a variable, the double % is replaced with a single % character, and the non-terminal %x is assigned to the variable. When the value of a variable is inserted into the output, it is re-interpreted, and any non-terminals found in the value will be replaced by new choices from the referenced productions. In practice, this feature will seldom be used. However, this feature allows one to simulate an arbitrary Turing machine with dgl variables. This implies that dgl grammars may be used to generate any kind of recursively enumerable data, not just context-free data. To illustrate, consider the non-context-free language a...ab...bc...c with an equal number of a's, b's, and c's. The following dgl grammar can be used to generate this language.

```
cc: variable;
main: "%a%{cc}";
a: a%ab%{c.cc},"";
c: c%{cc};
```

This grammar makes use of the fact that the initial value of a variable is the null string. It is possible to initialize a variable by placing the initial value after the "variable" keyword. The rules for specifying the initial value are identical to those for constructing the alternatives of ordinary productions. If more than one initial value is specified, all but the last will be silently ignored.

The rules for variable-assignment non-terminals are similar to those for other non-terminals. If the non-terminal contains a period, all characters after the first period are treated as the variable name. All characters preceding the first period are assumed to be the name of the production from which to choose a value for the variable. If the string before the first period is not the name of a production, the string itself will be assigned to the variable. If the string following the period is not the name of a variable, the entire nonterminal will be output, minus the % sign and the curly braces. The null string can be assigned to a variable by beginning a non-terminal with a period as in %{.x}. The non-terminal %{x.} is considered to be the same as %{x}.

Although dgl variables are universally powerful, they are not necessarily convenient for all purposes. Therefore, dgl provides three other types of variables: stacks, queues, rqueues, and hash_tables. All four types of variables can be thought of as dynamic productions that can change alternatives at run time. Variables have a single alternative, while stacks, queues, and hash_tables have several. Stacks and queues are similar to the "sequence" production, hash_tables are similar to ordinary productions, and rqueues are similar to “unique” productions. The alternatives for stacks, queues and rqueues are used only once and then discarded, while the alternatives for variables and hash_tables may be used many times. For queues, the alternatives are used in the order they were assigned, while for stacks the alternatives are used in reverse order of assignment. For hash_tables and rqueues, the choices are used in random order. An out-of-choices option can NOT be used with stacks, queues or rqueues. When a hash_table or an rqueue is referenced, all alternatives are chosen with equal probability.

Assignments are made to hash_tables, stacks, queues, and rqueues in the same way as assignments are made to variables. It is possible to specify initial values for stacks, queues, rqueues, and hash_tables by placing a list of values after the defining keyword. The rules for constructing this list are the same as those for constructing the alternatives of unweighted productions. An example of hash_table, stack, and queue declarations is given below.

```
a: stack;
b: queue;
c: hash_table;
d: stack a,b,c;
e: queue a,b,c;
t: rqueue q,e,d;
f: hash_table a,b,c;
```

For the above declarations, the first value selected from "d" or "e" will be "a", assuming no assignments have been made before the first selection. Strings containing non-terminals can be assigned to stacks, queues, rqueues and hash_tables by using a double % in the assigned string. When strings are selected from stacks, queues, rqueues, and hash_tables, they are reinterpreted.

9. Creating a Data Generator.

To create a data generator, one must first have access to the "dgl" program which is the dgl compiler. Source code and installation instructions may be obtained from the author. Once the dgl compiler has been installed, it can be used to create a data generator from a collection of dgl productions. The compiler runs under the UNIX operating system, and will run without modification on every version of UNIX known to the author, including System V and Berkeley 4.3.

The first step is to create a file containing dgl productions. This file should have the suffix ".dgl", and should contain a production named "main". The dgl compiler is used to transform the set of productions into a C program using the following command.

```
dgl  <specs.dgl  >specs.c
```

Error messages will appear on stdout. In most cases if error messages appear, the output file (specs.c in this example) will be empty. In any case, the output file will be unusable if error messages appear. The output of the dgl compiler must be compiled using the C compiler as demonstrated below. The C compiler must be capable of handling external names longer than 8 characters.

```
cc specs.c -o specs
```

The "specs" program is now the data generator specified by the productions of "specs.dgl." Each time the "specs" program is executed, it will make 100 selections from the "main" productions. This can be changed by putting some other number on the command line, as illustrated below.

```
specs 20
```

In this case, 20 selections will be made from the main production. Assuming that only ordinary productions have been used to construct the data-generator, the "specs" program will produce a different collection of data items each time it is invoked. This feature is implemented by having the data generator write its random-number seed to a file after each invocation, and reading that same file at the beginning of each new invocation. The file is created in the directory in which the data generator is invoked, so changing directories circumvents this feature. The name of the file containing the seed is <command.name>.rand, where <command name> is the name of the UNIX command used to invoke the data generator. For the "specs" example, the seed file will be named "specs.rand". The file name is determined dynamically, so changing the name of the program, or linking it to a new name, will change the name of the file.

10. Configuration Statements.

Dgl provides features for changing many of the things discussed in previous section. The state of systematic productions can be saved across invocations, the default number of selections made from the "main" production can be changed, the name of the random-number seed file can be changed or eliminated entirely, and one can prevent the user from changing the number of "main" selections at run time. Data generators may be linked with separately compiled code, and may be used as subroutines by other programs. These features should allow you to create a data generator that precisely fits your application.

Normally the data generator makes one or more selections from the production "main". If the name "main" is inconvenient for some reason, the "start" statement can be used to change the name of the starting production. The following statement changes the starting production from "main" to "S".

```
start: S;
```

If it is inconvenient to use the flag character "%" as the first character of each non-terminal, the "flag" statement can be used to change the leading character to something else. The following statement causes "$" to be the character that identifies non-terminals.

```
flag: $;
```

If one of the "illegal" characters is used as a flag, it must be enclosed in quotes as follows. (It will also be necessary to enclose all non-terminals in quotes.)

```
flag: "!";
```

If there is more than one "flag" statement, all but the last will be ignored. All productions in a particular grammar must have the same flag character.

Many dgl productions have an internal state that can be saved across invocations of the data generator. The types of productions mentioned so far that have internal states are "unique, "variable," "stack," "queue," "rqueue," "hash_table," "sequence," "counter," "head," and "chain." The "save" statement causes the internal state of named productions to be saved in the seed file. The following statement causes the state of production "a" to be saved across invocations.

```
save: a;
```

A list of production names can be specified as follows.

```
save: a,b,c;
```

If it is necessary to save the state of all productions, the following statement should be used.

```
save: all;
```

If it is necessary to save the state of all productions *except* a, the following two statements should be used

```
save:all;
nosave: a;
```

The rules for "save" and "nosave" are identical. The save and nosave statements can also be used to prevent the saving of the random-number seed. To prevent the seed from being saved across invocations, use the following statement.

```
nosave: seed;
```

The "save: all;" and "nosave: all;" statements normally do not affect the random-number seed. You can change this by explicitly declaring the seed using a statement similar to the following.

```
x: seed;
```

This statement allows "x" to be used in place of "seed" in save and nosave statements, and causes "save: all;" and "nosave: all;" statements to be applied to the seed as well as to other statements. When the seed is explicitly declared, it can be given an initial value. To understand how this works, you should be familiar with the UNIX documentation for "drand48". The dgl random number function is nrand48 found in the documentation for drand48. The random number seed is specified as three consecutive signed 16-bit integers. These 16-bit quantities are concatenated to form the 48-bit integers use by nrand48. (See the UNIX manual.) The initial value of the seed is specified as follows.

```
a: seed 16000,-3100,14001;
```

The initial value is used only when there is no seed file from the previous run in the current directory. If an initial value is not specified, a value of 4368, 2391, 1031 will be used. An initial value consists of three numbers separated by commas. The numbers must be in the range [-32768,32767]. Specifying an initial value for the seed also causes it to be affected by "save: all;" and "nosave: all;" statements.

The default number of selections from the "main" production is 100. The "repeat" statement can be used to change this value. The following "repeat" statement makes the default number of selections from the "main" production equal to 1.

```
repeat: 1;
```

The "options" statement can be used to prevent the user from changing the number of selections at run time. This statement is illustrated below.

```
options: nocount;
```

When the "nocount" option is specified, the default number of repetitions will be used for all invocations of the data generator, regardless of the value of any argument specified on the command line.

The name of the seed file is normally determined at run time by concatenating the string ".rand" to the name used to invoke the program. The name of the seed file can be changed using the "file" statement. The following statement changes the seed file name to "xyz.file".

```
file: xyz.file;
```

Note that *no suffix* will be appended to the specified name. In this case, the file name will be relative to the directory in which the data generator is invoked. One can specify a full path name to override this feature, as in the following example.

```
file: "/usr/lib/seed.file";
```

When a full path name is used, all users of the program use the same seed file. No special protection for the seed file is taken in this case. It is necessary to insure that the proper UNIX permissions and protections are set properly if the program is to be used by several different id's. It will normally be necessary to enclose the name of the seed file in quotes, because the slash (/) is one of DGL's special characters.

The first line of the seed file is a time-stamp. This is the date and time that the "dgl" compiler was run to create the data generator. If the time stamp of the seed file does not match the internal time stamp of the data generator, this implies that the seed file was created before the current data generator was compiled. Since recompilation can change the definition of certain non-terminals, and also change the structure of the grammar to the point where the seed file is no longer valid, a difference in time stamps causes the seed file to be ignored, and a new seed file to be created. The "ignorets" option can be used to circumvent this feature and force the reuse of the seed file regardless of the time-stamp. The "ignorets" option is specified as follows.

```
options: ignorets;
```

When the "ignorets" option is used, it is the user's responsibility to delete the seed file when it is no longer valid.

11. Action Routines.

Because the "variable" construct is universal, there is no need for dgl to include attributes and guarded productions. At times, however, it may be convenient to have action routines attached to certain productions. An action routine could be used, for example, to create a separate file of expected results for tests generated by the dgl productions. Since the underlying implementation language of dgl is C, action routines will normally be coded in C. There is also a feature that allows action routines to be separately compiled, which allows other languages to be used. The following construct is used to define an action routine.

```
a: action
(
    Arbitrary C Code
);
```

Each action routine will be turned into a separate subroutine, so the first statements should be the declarations of required local variables. Once an action routine has been defined, it may be referenced as follows.

```
ordinary: %a;
```

Action routines are allowed to produce output by calling the "interpret" subroutine. This subroutine must be invoked with a single character-pointer argument that points to a null-terminated string. If the string contains non-terminals, they will be replaced by choices from the referenced production in the output. The subroutine "interp2" can be used to interpret an arbitrary sequence of characters, which can possibly contain null characters. This subroutine requires two parameters, a character pointer which points to the first character of the string, and a count giving the number of characters to output. The count must be specified second. The string passed to "interp2" will be scanned for non-terminals. Neither "interpret" nor "interp2" make any changes in their argument string.

Uninterpreted output can be produced by calling the two routines "outcc" and "outstr". The first of these takes a single character as an argument, while the second takes a pointer to a null terminated string. All output passed to these routines (and the interpreter routines) will be redirected into a variable if a variable assignment is in progress when they are invoked. It is possible to circumvent variable assignments by writing to "stdout" directly, but this practice is not encouraged.

Action routines may produce error messages on "stderr" and may open and close their own files. They are also permitted to read "stdin" and write to "stdout," but writes to stdout must be done cautiously, because this is where the output of the data generator will appear.

There are four additional statements that can be used to include supporting code for action routines. The "defines" statement can be used to include any required "#define" statements and global variable declarations. The "initialize" statement can be used to include global initialization code, such as opening files. The "termination" statement can be used to include global termination code such as closing files. The "subroutines" statement can be used to specify globally accessible subroutines. Subroutines may also be separately compiled. The format of each of these statements is given below.

```
defines
(
    Arbitrary C code.
);
initialization
(
    Arbitrary C code
);
termination
(
    Arbitrary C code
);
subroutines
(
    Arbitrary C code
);
```

The initialization and termination statements will be turned into single subroutines. The dgl compiler will add the subroutine name and the enclosing curly braces. The first portion of the bodies of these statements should be declarations of local variables. The bodies of the other two statements are copied into the generated program intact. Note that if the "defines" statement contains "#define" statements, these must begin on the first character of a line.

It is possible for action routines, and the initialization and termination routines, to be separately compiled. A separately compiled action routine is declared by the following statement.

```
a: action;
```

The name of the separately compiled subroutine must be "a_select," for this declaration. In general, the string "_select" is appended to the name of the action routine to create the name of the separately compiled subroutine.

Separately compiled initialization and termination routines are declared by the following statements.

```
initialization: initl;
termination: terml;
```

For these two statements, the name of the separately compiled subroutines are "initl" and "terml" respectively. In general, the initialization and termination statements give the name of the separately compiled subroutine exactly.

It is possible for action routines and code included in "initialization" "termination" and "subroutines" statements to access command line arguments. The dgl compiler initializes two global variables "argc" and "argv". The main routine copies the arguments to "main" into these variables. These variables can be used in the same manner as normal "main" program arguments.

Action routines can access the state of various productions, and the tables used to hold the alternatives of various productions. See Section 18 for details on how to access the data structures of productions.

Action routines can be used to implement attributes and conditional interpretation of productions based on the values of attributes. When designing a grammar this way, remember that dgl interpretation is strictly right-to-left. Consider, for example, the following grammar, which uses the attribute "count" to generate a non-context free language.

```
defines(int count=0;);
main: %{s}%{t}\n;
s: "",a%sb%{a1};
a1: action (count++);
t: action (if (count>0) interpret("c%t););
```

12. State Variables.

At times it may be convenient to reset the state of a production to its initial value before exhausting all available choices. Dgl does not provide a specific mechanism for doing this, but instead provides a general mechanism that allows this to be done as a special case. A state_variable can be used to store the internal state of a production, and restore the state at some later time. Saving the initial state of a production in a state_variable allows the initial state to b e restored at any convenient time. A state_variable is declared using a statement similar to the following.

```
restore_s: state_variable;
```

An assignment to a state_variable is identical to an assignment to an ordinary variable, therefore, the non-terminal %{s.restore_s} assigns the current state of the production "s" to the state_variable "restore_s." To restore the state of "s", the non-terminal %{restore_s} is used. Neither of these non-terminals produces any output. Assigning the state of a production to a state_variable does not cause a selection to be made from the production, so the internal state of the production does not change. If you attempt to assign the state of a "stateless" production to a state_variable, the state_variable will be set to its initial "null" state. A reference to a "null" state_variable is ignored.

The association between a state_variable and the production whose state it contains is mad at run time, so a state_variable may be used to hold the state of several different productions, one at a time. A state_stack or a state_queue can be used to store the state of several productions at once. When the state_stack or state_queue is referenced, the state of *one* production is restored. For state_stacks, the states are restored in reverse order of assignment. For state_queues, states are restored in the order of assignment.

It is possible to save the state of the random number seed in a state_variable by declaring the seed and using the name in an assignment. This technique can be used to reproduce exactly a sequence of random choices.

The states of the following types of productions can be saved in a state_variable: unique, counter, sequence, chain, variable, stack, queue, hash_table, rqueue, state_variable, state_stack, state_queue, and seed. It is possible to save the state of a state_variable in itself. Assigning the state of one state_variable to another *does not* copy the value of the state_variable. For example, if "s" is a "sequence" and "v" and "w" are state_variables, the sequence %{s.v}%{v.w}...%w will cause the state of "v" to be restored, the state of "s" will remain unchanged.

The state of a state_variable (or state_stack or state_queue) may be saved across invocations of the data-generator by specifying the name of the variable in a "save" statement. The "save: all;" statement applies to state_variables, state_stacks, and state_queues.

13. Data-Generation Subroutines.

Normally a set of dgl productions is used to create a stand-alone data-generator. It is also possible to use the productions to create a data-generation subroutine. This is done by including the following statement in a set of dgl specifications.

```
options: subroutine;
```

When this option is specified, the dgl productions will be turned into a subroutine that can be linked with a main routine. In fact, three subroutines are created, an initialization subroutine, a termination subroutine, and a data-generation subroutine. The initialization subroutine must be called before the first invocation of the data-generation subroutine to initialize the random-number seed, and process the seed file. The termination routine must be called after the last invocation of the data-generation subroutine to write the seed file. Any declared initialization or termination code is executed by the initialization and termination routines.

The data-generation subroutine is invoked without arguments and returns a pointer to a character string. This string will be null-terminated, and will be the result of making *one* selection from the "main" production. The "repeat" statement and the "nocount" option are ignored when a subroutine is generated. The string returned by the data-generation subroutine will have been allocated using the "malloc" subroutine, and must be freed by the caller of the data-generation subroutine.

By default the name of the data-generation subroutine is "dgl" while the names of the initialization and termination routines are "dgli" and "dglt" respectively. You can change this by using the "name" statement as illustrated below. This statement changes the data-generation, initialization, and termination subroutine names to "gen", "init", and "term" respectively.

```
name: gen,init,term;
```

You can omit names from this statement if you don't want to change them all. If you change the initialization subroutine name but don't change the data-generation subroutine name, you must specify a leading command to indicate that the data-generation name is omitted. Similarly, if the termination routine name is specified, and the initialization routine name is omitted, two consecutive commas must be specified. Trailing commas should never be specified.

It is possible to include more than one data generator in a program. To do this it is necessary to guard against generating "duplicate definition" messages from the linkage editor. This problem can be avoided by specifying the "static" option. This option causes everything that does not need to be truly global to be specified as "static." The "static" option is specified as below. Note, however, that if this option is used, the output routines, and the data structures of productions will be inaccessible to separately compiled action routines. (Action routines that are specified in the grammar are unaffected.)

```
options: subroutine,static;
```

Because the "main" routine is no longer generated by the dgl compiler, it is impossible for the initialization routine to create the seed-file name from command-line argument zero. Because of this, the initialization and termination routines must be explicitly informed of the name of the seed file. If a "file" statement is specified, the initialization and termination routines will use this name. If no "file" statement is specified, the initialization and termination routines must be invoked with a single character-pointer argument that points to a null terminated string containing the name of the seed file. It is not necessary to specify the same name to both subroutines, but in this case it is the user's responsibility to pass the file along to future invocations of the program.

14. Arithmetic Expressions.

Expression productions can be used to generate numeric data that is difficult to generate by other means. For example the following grammar generates a series of perfect squares.

```
main: %s\n;
s: %{square}%{increment.var};
square: exp %{var} * %{var};
increment: exp %{var} + 1;
var: variable 0;
```

An expression production must contain the "exp" keyword following the production name. Arithmetic operations are performed using the operators "+," "-," "*," and "/." These multiplication and division have equal precedence as do addition and subtraction. Multiplication and division, however have precedence over addition and subtraction. Parentheses may be used to alter the usual precedence as in the following example.

```
k: exp ( %{a} + %{b} ) * ( 10 + %{c} );
```

The operands consist of single strings. Non-terminals are permissible, but concatenation, character sets and macros should not be used. (Only the implied concatenation operator is illegal, so 1 2 3 4 5 is illegal, but 12%{a}34 is legal.) The operands should generate strings of digits, however strings that contain characters other than [0-9] will be treated as zeros (if the first digit is non numeric) or as a number followed by a comment. That is, if a generated string consists of a sequence of digits followed by non-numeric characters, the digits up to the first non-numeric character will be treated as a number, and the remaining characters will be silently ignored. Parentheses may be nested arbitrarily deeply.

The value produced by an "exp" production will be a string of digits taken from the set [0-9]. If the result of the arithmetic operations is negative the string will begin with a dash ("-"). If the result is zero, the string will consist of the single character "0". In all other cases, the string will begin with a non-zero digit taken from the set [1-9].

15. Binary Output.

The output of several types of productions can be changed from an ASCII string to a 4, 2, or 1-byte binary number by specifying the “binary” keyword. For example the output of the following “range” production will be in binary.

```
a:   binary range 10,20;
```

Counter output may also be in binary as illustrated by the following production.

```
b:   binary counter 10,100,5;
```

The output of an ordinary production can be coerced to binary format as follows.

```
c:   binary     1,2,3,4,5;
```

When the “binary” keyword is used in an ordinary production, the number is converted to a binary integer using the following rules. First, if the number begins with “0x” it is considered to be a hexadecimal number, and may contain any of the characters 0-9, a-f, and A-F. If the number begins with 0 not followed by an x, it is considered to be octal and may contain any of the digits 0-7. If the number begins with 1-9 it is considered to be decimal, and may contain any of the digits 0-9. If the string contains any illegal digits, depending on format, the number is assumed to terminate just before the first illegal digit. Strings beginning with an illegal digit are coerced to a binary zero. Leading white-space characters (space, tab, or newline) are ignored. Negative numbers can be specified with a leading dash. (See the documentation for “strtol” on your system for specific details on the conversion which may differ from the preceding.) A “binary” production *must not* reference any other binary production either directly or indirectly.

By default the binary productions return a 32-bit value (a C long integer), however this can be changed by specifying a bit-size in parenthesis following the binary keyword as in the following examples.

```
a:  binary(8) range 10,20;
b:  binary(16) counter 10,100,5;
c:  binary(32)    1,2,3,4,5;
```

Any number may be used, however numbers less than 8 are rounded up to 8, numbers from 9-15 are rounded up to 16, and all other numbers, including those greater than 32, are treated as 32.

It is possible to specify portions of a bit-field by using the binary keyword with weighted productions. When weights are used with the “binary” keyword, the weight specifies the width of a subfield of the binary integer, rather than a probability. Fields are placed in the binary integer from left to right in the order specified. Fields that don’t fit into the integer are ignored. Unspecified bits will be set to zero. The following specifies a 32-bit integer with a 5-bit field, a 4-bit field, and a 3-bit field. The high-order bits hold the 5-bit field, which is followed by the 4-bit field. The 3-bit field follows the 4-bit field. The 20 low-order bits of the integer will be set to zero. The string generated by each field is converted to a 32-bit integer using the rules specified above. The integer is then truncated on the left to fit the field size.

```
d:  binary  5:(1,2,3,4,5,17),4:(6,7,8,9),3:(7,6,5,4,3);
```

Ordinarily the output of a binary production (including ranges and counters) is output as 1, 2, or 4 characters, just as if it were a string of that length. However if the string is assigned to a variable, the data generator will attempt to align 2 and 4 byte binary numbers on the proper boundary. (This may not work correctly on your machine. Check the generated code to make sure.) The first character of the string is assumed to be aligned to a 4-byte boundary. The alignment rule also automatically applies when a data generation subroutine is being used, since the output of a data generation subroutine is accumulated in an unnamed variable before being passed to the calling program.

When a data-generation subroutine is being used, it is possible to include complex data structures, such as linked-lists and trees in the output by using the “pointer” production. A “pointer” production is an ordinary unweighted production containing the “pointer” keyword, as illustrated below.

```
e:  pointer    “when”,“in”,“the”,“course”,“of”,“human”,“events”;
```

After a string is generated by such a production, instead of being output in the usual manner, the string is stored in a block of storage obtained using the “malloc” function, and a pointer to the storage area string is output. A pointer production may refer to binary productions, but binary productions may not reference pointers either directly or indirectly. Obviously, this type of production is of no use unless a data-generation subroutine is being used. Pointers are aligned as if they were 4-byte integers.

DGL also provides two types of floating point productions, which are illustrated below.

```
f:  float    1.2,2.1,1.1;
g:  double  3.3,4.4,5.5;
```

The first type of production produces a single-precision binary floating-point number, while the second produces a double-precision binary floating-point number. When alignment is performed, these numbers are aligned as if they were 32-bit integers. The string generated by these productions is coerced to a floating point number using the atof function. (See your local system documentation for details on what this function does on your system.) At the least, the following types of floating point numbers should be acceptable, in all their various combinations. Many systems also provide the ability to specify IEEE standard NaN’s and Infinities.

```
100
100.001
100.
.001
+100
-100
100e2
100e-2
1.1E+5
.001E-7
```

Pointers may reference any type of production including binary and floating point productions, but binary and floating point productions must not reference binary, floating point, or pointer productions, since these productions will create strings containing characters that have no meaning to the atof and strtol functions.

16. Experimental Features.

This section describes features of dgl that are experimental or still under development. It also describes certain features that were added to support research not directly related to test generation.

a. Permutations.

The first feature allows permutations of a list of elements to be generated systematically. An example of a permutation declaration is given below. This production generates all permutations of the letters a, b, c, d, e, and f.

```
exam: permutation a,b,c,d,e,f;
```

Successive references to this production will generate the strings "abcdef," abcdfe," "abcfde," and so forth. The rules for declaring a "permutation" construct are identical to those for defining an ordinary unweighted production. The state of a "permutation" construct can be saved across invocations using the "save" statement, and it is possible to save the state of a "permutation" construct in a state_variable, state_stack, or state_queue.

The permutation production may have an end-of-choices option following the "permutation" keyword.

b. Combinations.

It is also possible to generate mathematical combinations of elements. This corresponds to the operation of taking "m" things "n" at a time. It *does not* correspond to the intuitive idea of generating all combinations of certain possibilities. The intuitive idea is implemented using "chain" productions. An example of a combination declaration that implements the notion of taking six things three at a time is given below.

```
exam2: combination(3) a,b,c,d,e,f;
```

This construct will generate the following strings: "abc," "abd," "abe," "abf," "bcd," and so forth. To take "m" things "n" at a time, it is necessary to construct a production containing "m" alternatives, and place the number "n" in parentheses following the "combination" keyword. The rules for constructing a combination production are identical to those for constructing an ordinary unweighted production. The state of a combination construct can be saved across invocations using the "save" statement. It is possible to save the state of a combination construct in a state_variable, state_queue, or state_stack. The combination production may have an end-of-choices option.

c. External Productions.

The "external" declaration allows the right-hand side of an ordinary unweighted production to be constructed at run time either by an action routine or by the caller of a data-generation subroutine. The following is an example of an "external" declaration.

```
a: external;
```

This declaration requires the user to provide two variables which may either be separately compiled or placed in a "defines" construct. The first variable is an integer (int) and must be named "a_size" (replace "a" with the name of your production). The second variable is a pointer to a pointer to a character (char **) and must be named "a_table". This variable points to an array of character pointers that must be at least as large as the value of the "a_size" variable. Each pointer in the array points to a null-terminated string containing one alternative for the production. The alternatives may contain non-terminals. The "external" construct is experimental and may be changed or removed in the future. The alternatives of an "external" production cannot be weighted.

d. Dynamic Probabilities.

Another experimental construct is the dynamic construct. An example of a dynamic construct is given below.

```
a: dynamic 1:x, 2:y, 1:z;
```

All of the alternatives in a dynamic construct must be weighted. The difference between a dynamic construct and an ordinary weighted construct is that the dgl compiler generates a subroutine for a dynamic construct that allows the weights to be changed dynamically. This subroutine will normally be invoked either by an action routine or by the caller of a data-generation subroutine. This construct is one method that is currently being researched to determine the best method for constructing productions with non-standard probability distributions.

The probability of a particular alternative will normally be changed by an action routine, but it could be changed by the initialization and termination routines, or by the caller of a data-generation subroutine. To change the probability of the first alternative of the production “a” above from 1, to 7, the following function call must be executed.

```
a_setp(0,7);
```

In general, the name of the function used to change the probability of a production is <name>_setp, where <name> is the name of the production. The first argument is the number of the alternative to be changed, while the second is the new value of the probability of that alternative. Alternatives are numbered from zero in the order specified.

The current probability of alternative number “x” can be obtained from the array <name>_p, where <name> is the name of the production. The array <name>_p *must not* be modified directly. The following shows how to move the current probability of the 3rd alternative of production “a” above into the variable “x”.

```
x = a_p[2];
```

e. Poisson Productions.

It is well known that there are some probability distributions that cannot be generated by a probabilistic grammar. One of these is the Poisson distribution. Since this distribution is important in simulating failures, and since dgl is intended to be used to inject random errors into the designs of microelectronic circuits, the "poisson" construct was explicitly designed to allow data items to be generated with the Poisson distribution. For background, consider the following weighted production.

```
f: 1:TAIL%f, 1:HEAD;
```

This production simulates the random experiment of flipping a coin until a head appears. Since the weights are equal, the coin is fair. One can make the coin unfair by altering the weights. Each high-level selection fro "F" can be considered to be an event. The events generated by this production obey the Gaussian distribution. The following production allows events to be generated according to the Poisson distribution.

```
p: poisson a%p,"";
```

The following assumptions are made about this production. Each occurrence of the letter "a" represents one event. Each string generated by this production represents a sequence of events that occur in a unit time interval. The production must have exactly two alternatives, the first of which represents an occurrence of the event being simulated, and the second of which represents the non-occurrence of the event. The first alternative mus contain either a direct or indirect recursive reference to the production. Only one such recursive reference should exist for each use of the production. The second alternative must contain no direct or indirect reference to the production. If these assumptions hold, the events simulated by the above production will obey the Poisson distribution with parameter 1 and time interval 1. The time interval is always fixed at 1, but a different parameter may be used by enclosing it in parentheses following the poisson keyword, as in the following example.

```
p: poisson(3) a%p,"";
```

The parameter may be any floating point number that is acceptable to the UNIX function "atof". If the number contains minus signs, the entire number must be enclosed in quotes. As stated above, the "poisson" feature is experimental.

17. White Space and Comments.

White space (spaces, tabs, and newlines) are acceptable anywhere within a dgl specification. White space appearing in a quoted string is part of the string, other white space is ignored. Newlines should not normally be included in a quoted string, since the "\n" sequence represents a newline. However, newlines inside of a quoted string *will* be accepted, but a warning message will be produced for each such character found. White space may not be embedded in a dgl keyword.

Comments begin with the two-character sequence "/*" and end with the two-character sequence "*/". Comments may contain newlines. If the sequence "/*" appears inside a quoted string or inside an action routine, subroutines, defines, initialization, or termination statement, it *will not* be treated as the beginning of a comment. Beware of using the sequence "/*" in an unquoted string since it will be treated as the beginning of a comment. Comments may appear anywhere except embedded in a dgl keyword.

Case is significant in production names and production alternatives, so the production name "S" is considered different from the production name "s". However, case is *not* significant in dgl keywords, so MACRO, macro, Macro, and MaCrO all represent the same keyword. A slight modification to the installation procedure is available that restricts keywords to lower case.

18. Programming Notes.

The following contains hints and tips for the DGL installer, maintainer, and the advanced DGL programmer.

a. How to Get a Copy of DGL.

Dgl is available in source code format through anonymous FTP from internet node pangolin.usf.edu (network address (131.247.1.30). The source code is to be found in the file /pub/faculty/maurer/dgl-source/[illegible]source.tar.Z This file must be transferred in binary mode. Use the following commands to recover the source code.

```
uncompress dgl.source.tar.Z
tar -xvf dgl.source.tar
```

(This should normally be done in a special DGL directory.)

If you cannot transfer in binary, or do not have uncompress/tar on your system the same files are available in the directory /pub/faculty/maurer/dgl-source/source.files In this case, you will have to transfer all files individually (and there are LOTS of them).

b. How to Install DGL.

First recover the source code as above. Then edit the file “Makefile” and change the “install” dependency to put the executable file in the proper bin directory. Then type “make”. DGL is pretty much plain-vanilla C, but there are a few exceptions. When aligning 2- and 4-bit binary data, the existing code may not work for your machine. Search for “#ifdef” to locate the places where important stuff is done, and modify to fit your machine’s alignment rules. (It may take some doing to discover them, but creating data structures that contain a character variable followed by a short or a long will usually tell you what you need to know. Put some recognizable pattern in the variables and then write the area out as a string of characters. Watch out for byte ordering.

The method DGL uses to create the timestamp may not match the system calls used by your operating system. If the code refuses to compile because of something to do with the time-stamp, look up how to do it on your operating system and change the system calls to fit your environment.

Please send me mail at maurer@usf.edu if you have problems with installation, so the next version WILL work correctly on your machine.

c. How to Restrict DGL Keywords to Lower Case.

By default, DGL keywords are case-independent, so MACRO, macro, and MaCrO, are all considered to be the same. To restrict DGL keywords to lower case, edit the file Makefile, and find the line that reads:

```
mkulwhat  STRING  dgltok.h  whatis  <whatis.data  >whatis.c
```

Delete the two characters “ul” so the line now reads as follows.

```
mkwhat  STRING  dgltok.h  whatis  <whatis.data  >whatis.c
```

That’s it. From now on, only the lower case version of the keywords will be recognized.

d. How to Change or Add DGL Keywords.

Dgl keywords are not recognized directly by the DGL lexical analyzer. For reasons of performance, a separate keyword-identification program is used. This program runs as a subroutine of the lexical analyzer. It is generated by the mkulwhat or mkwhat program (See Section c above) using the file “whatis.data”. In this file all empty lines and all lines beginning with “#” are ignored. The other lines define DGL keywords, one definition per line. The format of these lines is as follows.

```
<keyword><tab><value>
```

There can be no white space on the line other than the separating tab. The tab MAY NOT be replaced by a space. The <keyword> defines the keyword as it appears in the source code, while the <value> determines the value that will be returned by the lexical analyzer. If the mkulwhat program is used to generate the identification program, all keywords should be lower-case, and the generated program will perform a case-independent test. If the mkwhat program is used, case is significant, and the keywords should have the proper case.

Normally the <value> field is a token that is defined in the dgl.yacc file. The values of these tokens are be available from the ".h" file generated by the yacc program. It is also possible to use numeric values here, but this is a dangerous practice, because these values may conflict with the values generated by yacc and may not conform to things that are recognizable in the yacc grammar.

No keyword may be defined more than once, but aliases for keywords can be created by using the same value more than once. For examples of this, see the "whatis.data" file.

e. How to Add a New Production Type.

The organization of this section leaves something to be desired, but it's certainly better than nothing.

e.1 Changes to the Parser.

Adding a new production type is reasonably straightforward, but involves several steps. The first step is deciding the form of the new production. The new production should have an identifying keyword as its first element. Next, it is necessary to define a yacc token to represent the keyword. Add the token name to the %token declaration in the dgl.yacc file. Then add the keyword definition to the whatis.data file using the instructions in Section d above.

Now, find the production "tail_spec: ..." in the dgl.yacc file. Add one line to this production. The form of this line should be as follows.

```
| <nonterminal> { production_handler() }
```

Note that the line begins with a long vertical line. The nonterminal is the start-symbol for the right-hand side of productions of the new type. The productions for this non-terminal must specify the new keyword as the first element, followed by pretty much whatever you want. The last thing specified by the production and its subordinate productions must be a semicolon, which is specified using the token SEMICOLON. The production must store data in global variables using action routines. The data will be used by the "production_handler()" routine to create the generated code for the production. If you want some standard things such as the body of an ordinary production, a string, or the body of a weighted production, existing productions should be used to recognize these items. The production "string_list" recognizes the body of an ordinary production, while the production "colon_list" recognizes the body of a weighted production. See the dgl.yacc file for other useful productions.

If your production must have an end-of-choices option, this option normally follows the keyword, and is specified by placing the nonterminal “eocopt” at the position where the end-of-choices option must be specified.

If you require new global variables to hold the data obtained from parsing your production, you must define them and initialize them in the file “dglgbls.c”. An “extern” declaration for each variable must be placed in the file “dglgbls.h”. This will permit the global variables to be accessed from any DGL compiler module.

If you wish, you may add a production alternative “<keyword> error” to handle reporting of format errors in productions of your type. This alternative should be followed by an action routine that calls yyerror. The routine yyerror will set a flag that prevents further generation of object code, and will report the number of the line upon which the error occurred. Yyerror must have one argument which is a pointer to a null terminated string. Currently all parser error messages are contained in the data structure “errmsgs” which is declared in a yacc escape in the dgl.yacc file. See dgl.yacc for examples of use.

e.2 Starting the Selection Routine.

The real work in adding a new production type is in creating the production handling routine. Your production handler must be a separate “.c” file (remember to include it in the Makefile) and must include stdio.h, and the local files “dgltok.h” and “dglgbls.h” in that order.

There are several actions that your routine MUST perform. The first statements in your production handler must be the following.

```
if (errfnd)
{
    return;
}
```

This will prevent your routine from generating code after a syntax error has occurred.

e.3 Handling the Production Body.

If your production contains the body of an ordinary or weighted production, the next steps you must perform are the following.

```
temp2 = xstkhead;
xstkhead = xstkhead->next;
```

This removes the parsed form of the production body from the string stack. The variable temp2 must be declared as follows.

```
XSTRUCT   *temp2;
```

In most cases, you will want to add the following statement.

```
temp = mkstarr(temp2,0);
```

The variable “temp” is declared as follows.

```
STARHDR  *temp;
```

When your production contains a the body of an ordinary production, this statement will create a list of strings from the stored specifications. If your production contains the body of a weighted production, use the following statement instead of the THREE statements listed above.

```
temp = mkparr(0);
```

The variable “temp” is declared as follows.

```
PROBHDR  *temp;
```

The function mkstarr returns a pointer to the following structure, which is declared in “dglgbls.h”.

```
struct starhdr
{
    int maxlen;                /* length of longest specified string */
    int entries;               /* number of strings after resolving sets & macros */
    char *refer;               /* ignore this field */
    struct staritem *itemhead;     /* pointer to list of strings */
}
```

The component itemhead points to a chain of structures of the following type.

```
struct  staritem
{
    struct staritem *next;          /* NULL if this is the last */
    int  length;                    /* length of item excluding trailing NULL char */
    char *data;                     /* pointer to null terminated string */
}
```

The type PROBHDR has the following format.

```
struct probhdr
{
     int  entries;      /* the number of items after resolving sets & macros */
                        /* Note, if things of the form 2:(a,b,c) are used, this */
                        /* will create ONE entry containing three strings */
     int total;         /* total of all probabilities.  2:(a,b,c) counts as 2, not 6. */
     struct probitem *itemhead;  /* pointer to a list of lists */
}
```

The component “itemhead” points to a linked list of structures of the following type.

```
struct probitem
{
     struct probitem *next;        /* NULL if this is the last item */
     int probval;                  /* the weight of this string (or list of strings) */
     struct  starhdr  *data;       /* see starhdr above */
}
```

e.4 Entering Your Production in the Hash Table.

Following the mkstarr, or mkparr call, your routine should perform the following call.

```
savid(nameval,<token>,<size or zero>,0,<width or zero>);
```

The variable “nameval” is a global variable (a character array) containing the name field of your production (the part that comes BEFORE the semicolon). The field <token> must be replaced by the yacc token which identifies the keyword of your production. The three remaining fields may be set to zero. The field <size or zero> is used by ordinary productions to record the number of entries from the starhdr structure, while the field <width or zero> is used by ordinary productions to record the maxlen field of the starhdr. You may use these fields for other purposes if you wish, but it is recommended to set them to zero. Without this call, the data generator will not be able to find the selection routines for productions of your type, so don’t omit it.

e.5 Terminating the Selection Routine.

If your production contains the body of an ordinary production, the last two statements in your routine must be as follows.

```
starfree(temp);
xfree(temp2);
```

If your production contains the body of a weighted production, replace these two statements with the following.

```
freeparr(temp);
```

e.6. The name of the Selection Routine.

The remainder of the production handler is used to generate code for your production. Remember that your production handler must generate a routine named <name>_select, where <name> is the null terminated string contained in the global variable “nameval”. This is the selection routine for your production. This routine is not permitted to have arguments. The code for this routine must be written using the FILE pointer “textfile”. DO NOT write anything to stdout. If you require any global variables, write their declarations using the FILE pointer “datafile”.

e.7 The End-of-Choices Option.

If you have specified an end of choices option for your production the value of the end-of-choices option is stored in the following global variables.

stop_spec	Set to 1 if STOP was specified.
continue_spec	Set to 1 if CONTINUE was specified.
abort_spec	Set to 1 if ABORT was specified.
restart_spec	Set to 1 if RESTART was specified.
dflt_spec	Set to 1 if no option was specified.
next_spec	Set to 1 if NEXT(<name>) was specified. The global variable “nextname” contains the name in the form of a null-terminated string.

The handling of the end-of-choices option is up to you, but if you wish to conform to the standards set by other productions, you should do the following. First, generate a global integer variable named <name>_next, where <name> is the name of your production. The declaration should initialize this variable to zero. The first part of your generated selection routine should test <name>_next for 1. If it is set to 1, generate the following code, depending on which end-of-choices option has been specified. This code should PRECEDE the code for making a selection.

STOP	Set the global variable “stop_run” to 1 and return.
CONTINUE	Return.
ABORT	Generate an error message on stderr, and call the C function “abort”.
RESTART	Reinitialize the production state to its initial state, and continue with the selection process.
NEXT	Execute the following statement interpret(“%{<nextname>}”) where <nextname> is the name specified in the global variable “nextname”. Place a return statement following the call to the interpreter.

For the NEXT option it is wise to call the interpreter rather than the production selection routine, because the production may be undefined. The interpreter will handle the undefined production properly.

Now, following the selection code, generate code to test whether the end-of-choices condition is present. If this condition is present, then set the global variable

<name>_next to 1. If the STOP option has been specified, it is also necessary to set the global variable “stop_run” to 1.

e.8 Generating Tables From Production Bodies.

If you want to create a tables out of an ordinary production body, the function “mkslst” is used for this purpose. A typical call is as follows.

```
mkslst(nameval,temp,-1)
```

The variable “temp” must be a pointer to a STARHDR and “nameval” must be a string, usually the name of the production. The function mkslst will generate an array of character pointers named “<nameval>_table”, where <nameval> is the string pointed to by the first argument.

If you want to create tables from a weighted production body, use the mkslist call on each STARHDR in the chain, but in this case you must use either the “nameval” field or the third argument of mkslst to create a different name for each table. If the third argument of mkslist is greater than or equal to zero, the argument will be converted to a 3-digit ascii number with leading zeros as appropriate, and will be appended to the name of the generated array. Thus the call mklst(“abc”,temp,1) will generate an array with the name “abc_table001”.

e.9 Saving the State of a Production Across Invocations.

If you want to save the state of your production across invocations of the data generator, you must provide code for writing the state of your production to the seed file, and for reading the state back in and restoring it. Processing of the seed file normally takes place AFTER any initialization operations for the production. The code for saving the states of productions is contained in a routine called “putstate” which is generated by the DGL routine “mkptst”. The code for reading the states of productions is contained in the routine “getstate” which is generated by the DGL routine “mkgtst”.

It is best to start with mkptst. Examine the existing routine and to get an idea about what is going on. The state of your production MUST start with the name of your production followed by a space. Following the space, on the same line, one normally places a set of values describing the state of the production. (Remember to save and restor the contents of the <name>_next variable if the production has an end-of-choices option.) It is necessary for you to end your values with a new-line character, so the next production state begins at the beginning of a line. Other than this, the rest of the line is free-form, although you may want to follow some simple rules to simplify reading the state back into memory. If you write out several strings, write them out with spaces separating them. If you write out numbers, convert them to ascii strings first. If you wish to write out a string containing spaces, non printable characters, OR the “$” character, use the following convention for writing strings. An arbitrary string begins with a $ followed by an ascii number which gives the length of the string. The ascii number must be followed by a space, and must immediately follow the $ without intervening spaces. The remainder of the string has a length equal to the ascii number and may contain ANY 8-bit character. The state of a variable is always written using this convention.

For reading the state of your production back into memory, the process can be simplified by using some of the predefined functions that will be automatically generated for you. Specifically, to obtain the next white-space delimited string, execute the following call.

```
ts = getstr(ifile,0);
```

The variable “ts” will contain NULL if no such string exists. Otherwise, ts will be a pointer to a string. The length of this string is limited to 200 characters, and it will be null terminated. The pointer actually points to a static work area, so the string must be duplicated if you need a string pointer in your initialization.

To read the length of a string, use the following call.

```
ts = getstr(ifile,1);
```

Note the difference in the second argument. This flag causes the initial $ to be stripped off of the string being read, but in all other respects, the behavior of the function is the same. To read the text of the string (which may be arbitrarily long), perform the following function call.

```
ts = getstr2(length,ifile);
```

The function getstr2 returns a pointer to a string that has been allocated using malloc. You should not count on this string being null terminated, and because it may contain several null characters which should be treated as data, use memcpy function calls to copy a specific number of characters, rather than using strcpy. If you no longer need the string returned by getstr2, you must free it using a “free” function call.

e.10 Saving the State in a State Variable.

There are several steps that must be taken if you want to save the state of your production in a state variable. You must, of course, add code to generate the save and restore operations for your type of production. In some cases, you will also need to generate code to copy the saved values from one state variable to another. Finally, you may need to alter the code used to read and write the state variable to the seed file. The value as recorded in the state variable is dynamically typed. Things will be simplified considerably if you can use one of the available types. (The types do not correspond one-for-one with production types.) The structure used to save the state of a production is as follows.

```
struct statev
{
    struct statev *next;
    char *name;
    int size;
    int vnxt;
    int type;
    union
    {
            char *c;
            int *i;
            STATEVX *x;
            struct statev *v;
            STAK *k;
            int ii;
            RSTAK *r;
    }
    val;
}
```

The current usage of these fields is as follows.

next	Used only for state stacks and state queues.
name	A character string containing the name of the production (malloc'ed).
size	Used primarily for variables -- number of chars in variable.
vnxt	Used for <name>_next variables.
type	See below.
val	Format depends on the "type" component.

Regardless of current conventions, you may use the "size" "val" and "vnxt" fields for whatever you choose. Currently, the following types are in use.

0 - The "val" field is unused. (COUNTER &SEQUENCE)
1 - The "val" field is a character string, whose length is in "size" (VARIABLE)
2 - The "val" field points to a malloc'ed array of integers, the size of which is one larger than the value of the "size" component. (UNIQUE,DYNAMIC)
3 - The "val" field points to a malloc'ed array of integers, the size of which is equal to the "size" component. (PERMUTATION,COMBINATION)
4 - Same as 2 (No longer in use, but don't play with it.)
5 - Same as 3 (No longer in use, but don't play with it.)
6 - The "val" field contains a pointer to a linked list of STATEVX structures. (No longer in use, but don't play with it.)
7 - The "val" field points to a linked list of strings, using the STAK structure for chaining. The "size" component contains the string count.
8 - The "val" field contains a pointer to another state-variable value.

9 - The “val” field contains a linked list of state-variable values.
10 - The “val” field contains an integer.
11 - The “val” field contains the state of a “chain” or “head” production.

If you add new types, use values of 1000 or greater to designate them. If you can adapt yourself to the existing types, then you need only modify the two routines mksvsv.c to save the state of your production in a state variable, and mksvrs.c to restore the state of your production from the state variable. These routines are pretty straightforward, so look at them for clues as to saving your state. (Steal code where possible.)

If you add a new type, you must also modify the mksvcp.c, mksvfr.c, mkprsvar.c, and mkrdsvar.c. The routine mksvcp.c is used when storing the state of one state variable in another, and in other arcane places. It must know about your new value so it can copy values of your type. The routine mksvfr.c is used to destroy the value of a state variable, and is called every time a state-variable is reassigned. Mkprsvar.c is used to write the value of a state variable to a seed file, while mkrdsvar.c is used to read the value of a state variable from the seed file. Look at these routines for examples of how things are done, but if you’ve gotten this far, you probably don’t need many hints.

f. How to Access Production Data Structures.

In this section, it is assumed that the name of the production is “abc” for all productions. Where you see “abc” replace with the name of the production you are interested in.

For all productions, abc_select() is the selection routine.
For variables, abc_assign(production_name,name_length) the assignment routine,
where “production_name” is the production from which the value should be selected.
IMPORTANT: for all variables named “abc_table” if the alternatives of the production
are all 1-character strings, the array of character pointers will be replaced by an array
of characters. (I think there are some productions for which this does not happen,
but I can’t remember which.) If the only alternative of the production is a macro
reference, “abc_table” will not exist. (A non-zero length specification is enough to
force the creation of “abc_table”.)

ACTION	NONE	
BINARY	abc_table	char * array containing pointers to all alternatives.
	abc_size	int containing the number of entries in abc_table.

CHAIN	abc_table	char * array containing pointers to all alternatives.
	abc_size	int containing the number of entries in abc_table.
	abc_value	RSTAK * containing current state NULL initial.
	abc_next	flag indicating whether EOC condition has occurred.
	rs_left	When chain active, production to left of current in binary parse tree.
	rs_above	When chain active, production above current in binary parse tree. (Both rs's NULL => no chain act)
COMBINATION	abc_table	char * array containing pointers to all alternatives.
	abc_size	int containing the number of entries in abc_table.
	abc_value	int array containing the characteristic vector of the next choice
	abc_next	flag indicating whether EOC condition has occurred.
COUNTER	abc_value	int containing the next value of the counter.
	abc_next	flag indicating whether EOC condition has occurred (may not be present if not needed.)
DOUBLE	abc_table	char * array containing pointers to all alternatives.
	abc_size	int containing the number of entries in abc_table.
EXPRESSION	NONE	
EXTERNAL	abc_table	char * array containing pointers to all alternatives.
	abc_size	int containing the number of entries in abc_table.
FLOAT	abc_table	char * array containing pointers to all alternatives.
	abc_size	int containing the number of entries in abc_table.
HASH_TABLE	abc_table	char * array containing pointers to all init time alternatives.
	abc_size	int containing total number of entries in abc_table and abc_value.
	abc_first	Left over from an earlier age (unused, init to 1).
	abc_xsize	The total number of values in abc_table.
	abc_value	A linked list of STAK structures containing the dynamically added alternatives.

HEAD	abc_table	char * array containing pointers to all alternatives.
	abc_size	int containing the number of entries in abc_table.
	abc_value	RSTAK * containing current state NULL initial.
	abc_next	flag indicating whether EOC condition has occurred.
	rs_left	When chain active, production to left of current in binary parse tree.
	rs_above	When chain active, production above current in binary parse tree. (Both rs's NULL => no chain act)
ordinary prod.	abc_table	char * array containing pointers to all alternatives.
	abc_size	int containing the number of entries in abc_table.
PERMUTATION	abc_table	char * array containing pointers to all alternatives.
	abc_size	int containing the number of entries in abc_table.
	abc_value	int array containing permutation of the array indices 0 through n-1 for generating the next output
	abc_next	flag indicating whether EOC condition has occurred.
POINTER.	abc_table	char * array containing pointers to all alternatives.
	abc_size	int containing the number of entries in abc_table.
POISSON.	abc_table	char * array containing pointers to all alternatives.
	abc_size	int containing the number of entries in abc_table.
	abc_N	See generated code.
	abc_DIV	See generated code.
	abc_PN	See generated code.
	abc_T	See generated code.
	abc_F	See generated code.
QUEUE	abc_value	STAK * pointing to head of queue.
	abc_tail	STAK * pointing to tail of queue.
	abc_size	int containing number of items in queue.
	abc_first	No longer used. Init to 1.
RANGE	none	
RQUEUE	abc_value	STAK * pointing to head of queue.
	abc_tail	STAK * pointing to tail of queue.
	abc_size	int containing number of items in queue.
	abc_first	No longer used. Init to 1.
SEED	none	(but 3 element short array xnum contains current seed whether or not a SEED production is declared.)

SEQUENCE	abc_table	char * array containing pointers to all alternatives.
	abc_size	int containing the number of entries in abc_table.
	abc_value	int containing index of next alternative (init to zero).
	abc_next	flag indicating whether EOC condition has occurred.
STACK	abc_value	STAK * pointing to top of stack.
	abc_size	int containing number of items in stack.
	abc_first	No longer used. Init to 1.
STATE_QUEUE	abc_value	STATEV pointer pointing to head of queue (NULL means empty)
	abc_value	STATEV pointer pointing to tail of queue
STATE_STACK	abc_value	STATEV pointer pointing to top of stack (NULL means empty)
STATE_VARIABLE	abc_value	STATEV pointer pointing to value (NULL means empty)
UNIQUE	abc_table	char * array containing pointers to all alternatives.
	abc_size	int containing the number of entries in abc_table.
	abc_counts	int array containing number of each item left to select
	abc_remain	int containing total of all counts in abc_counts.
	abc_next	flag indicating whether EOC condition has occurred.
VARIABLE	abc_value	char * pointing to non-null terminated string.
	abc_length	int containing number of characters in string.
weighted prod.	abc_tablennn	char * array containing pointers to alternatives.
	abc_sizennn	int containing the number of entries in abc_tablennn.
		Note: one table is generated for each weight specified. The table contains the alternatives to which that weight belong. nnn is a 3-digit number identifying the table, numbers start with 000.

19. Conclusions.

Dgl provides an effective means of generating test data. Although many of the features of dgl are firm, it is a research tool that is supporting active research in VLSI testing and design verification. Because of this, dgl is currently under development and will probably remain so for some time. Some of the current features may disappear, others will be replaced by more useful features. More information about dgl can be obtained from the author.

20. Dgl Keywords and Reserved Characters.

The following is an alphabetical list of dgl keywords.

abort	hash_table	restart
action	head	rqueue
all	ignorets	save
binary	initialization	seed
chain	macro	sequence
combination	name	stack
continue	names	start
counter	next	state_queue
debug	nocount	state_stack
define	nodebug	state_variable
defines	nosave	static
double	option	stop
dynamic	options	subroutine
exp	permutation	subroutines
expression	pointer	termination
external	poisson	unique
file	queue	variable
flag	range	width
float	repeat	

The following is a list of all dgl reserved characters (sometimes called "illegal" in the text)

'	(	-
"	)	/
:	!	*
[	,	+
]	;	

INDEX