The Simple Language Generator: Encoding complex languages with simple grammars

Douglas L. T. Rohde

September, 1999

CMU-CS-99-123

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper introduces the design and use of the Simple Language Generator (SLG). SLG allows the user to construct small but interesting stochastic context-free languages with relative ease. Although context-free grammars are convenient for representing natural language syntax, they do not easily support the semantic and pragmatic constraints that make certain combinations of words or structures more likely than others. Context-free grammars for languages involving many interacting constraints can become extremely complex and cannot reasonably be written by hand. SLG allows the basic syntax of a grammar to be specified in context-free form and constraints to be applied atop this framework in a relatively natural fashion. This combination of grammar and constraints is then converted into a standard stochastic context-free grammar for use in generating sentences or in making context dependent likelihood predictions of the sequence of words in a sentence.

This research was partially supported by NIMH Program Project Grant MH47566 Part 1 (M. Seidenberg, M. MacDonald, D. Plaut co-PIs; J. McClelland, PD) and an NSF Graduate Fellowship. Correspondence may be sent to Douglas Rohde (dr@cs.cmu.edu), School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213–3890, USA.

Keywords: context free grammar, language generation, Carnegie Mellon University, SLG

1 Introduction

A common goal in the field of machine learning is the development of models that are able to capture the structure of a language, be it a natural, human language or something more abstract. For example, one might wish to learn the rules of grammaticality or use the language to comprehend and produce messages. Although it is often desirable to work directly with a complete natural language, in studying the behavior of a particular learning method or in comparing multiple strategies it is sometimes necessary to have at our disposal languages with well-understood and easily controlled properties.

To produce such a language, we typically rely on a grammar which defines the legal strings, or sentences, it contains. A *generative* grammar is one that can produce those sentences. With most reasonable languages, it is usually not very difficult to write a program to generate the language. However, one of the goals of many researchers is to train a neural network, hidden markov model, or other learning method to predict each word in a sentence. In order to evaluate such a model, we would need the theoretically correct predictions. Although there are many ways to generate a language, most of them do not enable the rapid calculation of word-by-word predictions based on the grammar.

A simple yet fairly powerful form of grammar is the context-free grammar, or CFG (see Hopcroft & Ullman, 1979, for an introduction). By specifying probabilities that each possible *production*, or transition, in the grammar is performed, we can control the distribution of sentences produced by a CFG. Thus we form a type of generative grammar known as a stochastic context-free grammar, or SCFG. The advantage of the SCFG is that it has been the subject of considerable analysis and we have reasonably fast algorithms for parsing and producing word predictions using SCFGs.

Unfortunately, if one is interested in designing complex languages by hand, the SCFG can be rather cumbersome. In order to produce a language of significant complexity, where *complexity* is used in a non-technical sense, the required grammar becomes long and complicated, involving considerable redundancy and a host of symbols, which often have rather abstract relationships to the final language. In the SCFG, probabilities must be specified for each production in the grammar, but reasonable probabilities are difficult to determine by hand if the symbols involved do not have clear mappings to well-understood properties of the intended language. Therefore, designing interesting SCFG grammars by hand becomes quite impossible.

The goal of SLG or the Simple Language Generator, is to allow a human to specify relatively concise and intuitive grammars which nevertheless define interesting languages. The grammar interpreted by SLG is similar in basic form to an SCFG, but it allows the designer to specify additional constraints that alter the resulting language. The ability to reuse constraints helps to eliminate redundancy. SLG can then convert the user's grammar to a standard SCFG. This process is known as *resolving* the grammar. Once we have obtained an SCFG which is equivalent to the original grammar, albeit much longer and more complex, we can easily generate sentences in the language or produce optimal word predictions.

This report explains the use of SLG and some of its inner workings. Section 2 describes the grammar specification language. Section 3 explains the process by which constraints are resolved. Section 4 explains the process of reducing or minimizing the size of grammars. Section 5 describes the method of converting grammars to Greibach Normal Form and how this is used to produce word predictions. Section 6 mentions some possible future extensions to the program and provides the address for downloading SLG. Finally, Appendix A explains the command-line arguments used to control SLG.

2 The grammar

The grammar interpreted by SLG[1] is a superset of a standard SCFG grammar. Therefore, any ordinary SCFG, and hence any finite state machine, can be handled in a straightforward manner. One can view the process of generating a sentence with an SCFG as the branching of an inverted tree. Each non-terminal symbol branches into the symbols in its chosen production. The grammar is context-free because the branching of each symbol depends only on the symbol itself and is unaffected by context, or the symbols around it. While CFGs are a convenient way to capture the syntax of many languages, and have thus attracted the attention of linguists, if we are concerned with the frequency of sentences, we must consider the semantics and pragmatics of natural languages, which play an important role in the choice of productions. However, it is not possible to introduce this type of information into an SCFG without restructuring it, which would destroy the nice, simple model of syntax.

[1] This report is based on SLG version 2.0.

What we would like to be able to do is to constrain the behavior of one symbol given the productions of one, or more, other symbols in the tree. For example, when producing natural sentences, we might want to constrain the choice of verb based the choice of subject or the choice of adjective based on the noun it is describing. Given a grammar and a list of such constraints, it is possible, although not entirely straightforward, to generate sentences that satisfy both the grammar and the constraints as long as the dependencies aren't circular. However, given such a model, it would be next to impossible to efficiently parse and generate word predictions. To do that, we would really like just a standard SCFG. But as long as the region of the tree affected by each constraint is contained, we should be able to eliminate the need for the constraints by restructuring just those portions of the grammar so that the constraint is effectively embedded in the context-free productions. This is the role of SLG.

Defining an SLG constraint involves two steps. First, a constraint function is defined which specifies how the choice of productions from the constraining symbol, or the *source*, affect the choice of productions of the constrained symbol, or the *goal*. Then the constraint must be applied to each appropriate pair of source and goal. This is done by specifying which sub-tree or sub-trees are affected by the constraint. The sub-tree begins at the *root*, or the symbol which is the lowest mutual-ancestor of the source and goal. Typically, noun-phrases and verb-phrases appear in many places in a natural language grammar. The ability to reuse a single subject-verb constraint function for each pair helps eliminate redundancy and segregate semantic/pragmatic information from syntactic information in the grammar. Because resolving a constraint only involves altering the paths through the tree that start at the root and extend to the source or goal, the use of constraints does not render the SLG grammar super-context-free in the theoretical sense.

2.1 Using the SLG grammar

```
      S  : NP VP "." |
           {LegalIntVerb, NP N, VP VI} |
           {LegalTrnVerb, NP N, VP VT};
      VP : VI | VT OP (0.7) |
 5         {LegalObject,  VT, OP N} |
           {LegalObject,  VT, OP N2};
      NP : the N;
      OP : the N | the N and the N2 |
           {DontRepeatObj, N, N2};
10    N | N2 : boy (0.3) | cat (0.3) | dog;
      VI : barked | slept;
      VT : bit | fed;

      LegalIntVerb {
15     boy | cat : slept;
        dog       : barked (0.8) | slept;
      }

      LegalTrnVerb {
20      dog | cat ! fed;
      }

      LegalObject {
        bit | fed : boy (0.6) | cat (0.2) | dog;
25      fed       ! boy;
      }

      DontRepeatObj {
        boy ! boy;
30      cat ! cat;
        dog ! dog;
      }
```

Figure 1: A grammar for producing simple sentences.

Figure 1 contains a sample SLG grammar, illustrating the syntax and many of the available features. The first 12 lines contain symbol definitions. A symbol definition begins with a list of the symbols to be defined, separated by | characters. It is convenient to read the | character as "or". A symbol name can consist of any string of characters excluding white space and the following special characters: ; | : , {} () ! Alternately, a symbol name can be any string enclosed in double quotes. This allows multi-word symbols and symbols using the special characters. The first symbol defined becomes the start symbol.

Each of the symbols in the definition list will receive the same definition. That is, they will have the same set of productions and will be the *root* symbol for the same sets of constraints. It may not seem particularly useful to have equivalent symbols, but it often comes in handy when one needs to apply different constraints to otherwise identical symbols, and it can sometimes help make the grammar more clear. An example of a shared definition is that of "N | N2" on line 10 of the grammar. Because we created two different nouns non-terminals, we can distinguish between them in the constraint on line 9.

Following the definition list is a colon and then a list of *productions* and *constraints*, separated by |'s. The definition is terminated with a semicolon. A production is a string of symbols (separated by whitespace) followed by an optional probability (prob.) enclosed in parentheses. The probabilities for all possible productions from a symbol must sum to 1.0. Any productions whose prob. is not specified will be given the same prob., which is calculated such that the overall sum becomes 1.0. Therefore, if three productions are defined and a prob. of 0.6 is specified for the first and no prob. is specified for the others, the other productions will default to 0.2.

Constraints are enclosed in curly braces and consist of three parts. The first specifies the constraint function, which must be defined separately. The second specifies the *source path* and the third specifies the *goal path*. When a sentence is generated with a CFG, we can view the process as the branching of a tree, beginning with the start symbol, which is S in this case. Figure 2 illustrates the parse tree for a sentence generated by the example grammar.

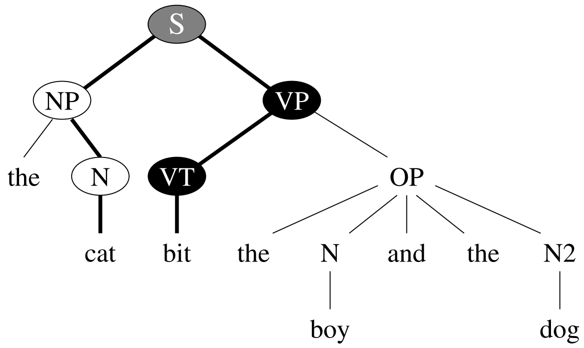

Figure 2: Parse tree from the grammar with a source path shown in white and a goal path shown in black.

When a constraint is given in a symbol definition, the symbol currently being defined is called the *root* of the constraint. For example, the root of the constraint on line 9 is OP. Note that the root of a constraint need not be the start symbol of the grammar. A constraint path consists of a series of symbols separated by whitespace. It matches a path through the states in the parse tree which begins at the root symbol. Each of the symbols in the path must match a symbol in the parse tree at the next level down. If either the source path or the goal path does not match a path in the tree, the constraint does not apply.

In the case of the tree in Figure 2, the constraint "{LegalTrnVerb, NP N, VP VT}", which has root S, is applicable. The source path is marked with white ovals and the goal path with black ovals. The last symbol in the source path is called the *source*, because it will be the source of the constraint. In this case, the source is N. The last symbol in the goal path, VT, is called the *goal*. A constraint is only valid if the first symbol on the source path is different from the first symbol on the goal path and the two symbols appear together in at least one root production. Additionally, the root symbol itself may not appear on the source or goal paths except as the first symbol.

The choice of production out of the source, or the production that the source symbol performs, will constrain the choice of production out of the goal. It does this using the specified constraint function, which must be defined separately. A function definition consists of the name of the function followed by a list of terms enclosed in curly braces. Each term begins with a list of productions called the *source list*. These must be valid productions out of the source, as given by the definition of the source symbol. Note that the elements in this list are *productions*, not just symbol names. In the current example, all of these productions happen to consist of a single symbol, but in general it is possible for constraints to involve more complex productions comprising a series of symbols. Following the source list is either a colon or exclamation point and then the *goal list*, which is a list of possible productions out of the goal symbol.

If the character separating the source and goal lists is an exclamation point, rather than a colon, then the goal list specifies productions which *cannot* be taken from the goal symbol if the source symbol produces one of the productions in the source list. This is a convenient way to eliminate selected productions. For example, the LegalTrnVerb function says that if the subject is dog or cat then the transitive verb cannot be fed. Probabilities should not be specified when using an exclamation point.

On the other hand, if the character separating the source and goal lists is a colon, the goal list specifies the only legal productions from the goal symbol if the constraint term applies. In addition, probabilities can be specified for the goal list productions that will modify the distribution of productions produced by the goal. The new distribution does not replace the old one. Rather, it *filters* the distribution. When one distribution filters another, the corresponding terms are multiplied and the results re-normalized. For example, symbol N normally produces boy, cat, or dog with respective probabilities of 0.3, 0.3, and 0.4. When filtered by the first term in the LegalObject function, which specifies a distribution of 0.6, 0.2, 0.2, the resulting distribution becomes 0.5625, 0.1875, 0.25. Note that if a production has prob. 0.0 in either distribution, it will have prob. 0.0 in the result. If no probabilities are specified in the goal list, the constraint will eliminate goal productions that are not listed but will not change the relative likelihood of the listed productions.

Although it is usual for each source production to appear in at most one term in a constraint function, it is sometimes useful to define multiple terms for a production. For example, you might have one term that represents animals and one that represents fierce things, both of which include dog. If a source production matches more than one term then the goal productions are filtered by each of the matching terms. The order in which filtering occurs does not matter.

Let us examine the constraints employed in the example grammar. LegalIntVerb is used to constrain the choices of intransitive verbs given the subject of the sentence. Note that the constraining path in this constraint, NP N, will always be matched because an S must produce an NP and an NP must produce an N. The constrained path, VP VI, may not be matched because a verb phrase may contain a transitive verb rather than an intransitive one. If the constrained path does match, then this constraint will affect the choices of intransitive verb based on the choice of noun for the subject.

The definition of the LegalIntVerb function specifies that if the subject is boy or cat then the intransitive verb must be slept. However, if the subject is dog, then the intransitive verb will be barked with prob. 0.8 and slept with prob. 0.2. The LegalTrnVerb function is used to constrain the possible transitive verbs given the subject. If the subject is either dog or cat, the transitive verb cannot be fed. Therefore, it must be bit. Note that boy is not listed in any of the terms in the definition of LegalTrnVerb. Therefore, if the subject is boy, the constraint has no effect and the transitive verb will be bit or fed with equal likelihood.

The LegalObject function is used to constrain the choices of object given the transitive verb. The first term says that if the verb is bit or fed, all nouns are possible, but their probabilities have been altered. The second term, however, specifies that if the verb is fed, the object cannot be boy. In this case the prob. of the object becoming cat will be $3/7$ and the prob. of dog will be $4/7$. Finally, the DontRepeatObj constraint eliminates the possibility of a compound object consisting of the same noun twice, such as "…the dog and the dog".

The amount or type of whitespace does not matter in the SLGgrammar file, except that any line starting with a # will be treated as a comment.

2.2 Other features

One feature that has not yet been mentioned is the use of constraint wild-cards. The symbol names in the source and goal path of a constraint may be replaced with a *. This will match any symbol and allows a single constraint to apply to several paths. When resolving the grammar, the constraint will actually be turned into a set of constraints in which the wild cards have been replaced by all possible combinations of symbol values.

Another useful feature is the *epsilon* production. A standard concept in CFGs, these are productions that generate nothing and thus eliminate the current symbol from the tree. While these do not alter the theoretical power of the grammar, their use can simplify grammars. Consider the two examples shown in Figure 3. These are equivalent grammars that produce a noun-phrase containing an optional article and optional adjective before the noun. The first grammar does not use epsilon productions and therefore must specify all four possible types of noun-phrase. The second grammar uses epsilon productions, which are written as an empty pair of double-quotes, to make the article and adjective optional while freeing the user from enumerating every possibility.

Finally, constraints may be given a priority which determines the order in which they will be resolved. By default, if there are no other dependencies, constraints are handled in some arbitrary order when resolving the grammar. However, sometimes the resolution process goes faster if certain constraints are resolved before others. The user can influence the order of resolution by giving constraints a priority. By default, constraints have priority 0, but the priority can be changed by placing a fourth field in the constraint specification, as in {foo, NP N, VP VT, 3}. The priority may

```
NP : N | ART N | ADJ N | ART ADJ N;
ART: the | a;
ADJ: green | putrid;
```

```
NP : ART ADJ N;
ART: "" | the | a;
ADJ: "" | green | putrid;
```

Figure 3: Using epsilon productions to simplify a grammar.

be negative. Higher priority constraints will be resolved first. For most grammars, this will have no noticeable effect and is most useful if you would like to observe the intermediate stages of the resolution process under a particular constraint ordering.

2.3 Limited cross-dependency

A main attraction to CFGs has been their ability to conveniently capture center-embedding, which is a common feature of English and most other languages. A nested center-embedded sentence might have the general structure N1 N2 N3 V3 V2 V1, where V1 depends on N1, V2 on N2, and so on. These are easily captured by CFGs. However, of considerable interest and trouble to the linguistic community has been the existence of a few languages, most notably Dutch and Swiss German, that permit cross-dependencies (Christiansen & Chater, in press), which have the general structure N1 N2 N3 V1 V2 V3. These cannot in general be described by a CFG and, even if the depth of the embedding is limited, are difficult to describe once agreement and semantic constraints are introduced.

```
S: N1 N2 N3 V1 V2 V3 |
   {N-V, N1, V1} | {N-V, N2, V2} | {N-V, N3, V3};
N1 : dog | dogs | cat | cats;
N2 | N3 : dog | dogs | cat | cats | "" (0.8);
V1 | V2 | V3 : barks | bark | purrs | purr | "";

N-V {dog:barks; dogs:bark; cat:purrs; cats:purr; "":"";}
```

```
S:  N NP V VP | {N-V, N, V} |
    {NP-VP, NP, VP} | {N-V, NP N, VP V} |
    {NP-VP, NP NP, VP VP} | {N-V, NP NP N, VP VP V};
N:  dog | dogs | cat | cats;
V:  barks | bark | purrs | purr;
NP: N NP | "" (0.8);
VP: V VP | "";

N-V {dog:barks; dogs:bark; cat:purrs; cats:purr;}
NP-VP {"":""; N NP:V VP;}
```

Figure 4: Two SLG grammars for limited cross-dependency.

However, cross-dependencies of limited depth are not too difficult to describe using an SLG grammar. Figure 4 shows two ways in which, in rather simplified form, one might write such a grammar. The first uses a flat representation to explicitly allow up to three possible pairs. Three constraints are required to implement agreement. The use of the epsilon production allows N2 and N3 to be optional, and the constraints prevent the corresponding verbs from appearing in the absence of the nouns. When resolved into an SCFG, this grammar requires 25 non-terminal symbols and 124 productions.

The second example uses a nested structure, which may be more convenient and more linguistically reasonable in a full grammar. In this case, two constraints are used for each possible depth of embedding. However, the grammar isn't entirely adequate since it could produce embeddings beyond depth two which would not be subject to the constraints. The maximum depth of embedding could be bounded by introducing a path constraint, as mentioned in Section 6.

```
S  : SP VI . (.25) | SP VT OP . |
  {sub-intr, SP NP N, VI} | {sub-trns, SP NP N, VT} |
  {trns-obj, VT, OP NP N} | {sub-obj, SP NP N, OP NP N} |
  {intrans-ref, VI, SP RC VI};

SP | OP : NP | NP RC (.3) |
  {sub-intr, NP N, RC VI} | {sub-trns, NP N, RC VT} |
  {trns-obj, RC VT2, NP N};

RC : who VI | who VT OP | who SP VT2 |
  {trns-obj, VT, OP NP N} | {sub-trns, SP NP N, VT2};

NP : ART ADJ N | {noun-art, N, ART} | {noun-adj, N, ADJ};

ART: "" | the | a;

ADJ: "" (0.6) | quick | happy | hungry | nasty | mangy | crazy | sleazy;

N  : boy | boys | girl | girls | Mary | John | cat | cats | dog | dogs;

VI : walks | walk | bites | bite | eats | eat | barks | bark;

VT | VT2 : chases | chase | feeds | feed | sees | see | walks | walk | bites |
  bite;

sub-intr {
  boy  | girl  | Mary  | John  : walks | eats;
  boys | girls : walk  | eat;
  cat  | dog   : walks | bites | eats  | barks;
  cats | dogs  : walk  | bite  | eat   | bark;
  cat  | cats  ! bark  | barks;
}

sub-trns {
  boy  | girl  | Mary   | John : chases | feeds | sees(.1) | walks;
  boys | girls : chase  | feed       | see(.1) | walk;
  cat  | dog   : chases | sees(.2)   | bites;
  cats | dogs  : chase  | see(.2)    | bite;
}

trns-obj {
  walk | walks : cat | cats | dog | dogs;
  see  | sees  : cat | cats;
}

sub-obj {
  Mary ! Mary;
  John ! John;
}

intrans-ref {
  walks | walk ! walks | walk;
  bites | bite ! bites | bite;
  eats  | eat  ! eats  | eat;
  barks | bark ! barks | bark;
}

noun-adj {
  boy  | boys | girl | girls | Mary | John ! mangy;
  John | cat  | cats | dog   | dogs ! sleazy;
}

noun-art {
  Mary | John : "";
  boys | girls | cats | dogs ! a;
  boy  | girl  | cat  | dog  ! "";
}
```

Figure 5: A more complex SLG grammar.

2.4 A larger example

Figure 5 shows a much more complex SLG grammar which produces some reasonably interesting English sentences. Once it is resolved, this grammar produces an SCFG with 140 non-terminals and 442 productions, which is considerably larger than the original grammar. Some sentences produced with this grammar are listed in Figure 6.

```
a dog bites the happy boys .
dogs who chase the hungry cat bite nasty girls who walk .
the crazy cat walks .
the hungry cats who walk chase the hungry dog who chases the crazy girls .
the mangy dogs who walk bite the quick dog .
a nasty cat who sees the mangy cats bites .
girls chase the sleazy girls .
hungry cats eat .
the nasty cats bite Mary .
the nasty cat who a crazy girl chases bites the crazy boys .
```

Figure 6: Sentences generated by the complex grammar.

3 Resolving the grammar

This section explains the process by which SLG takes a grammar involving constraints and transforms it into a grammar in standard SCFG form. It does this by *resolving* each of the constraints. Resolving a constraint is a rather complex process, but essentially involves splitting each of the symbols along the source and goal paths into sub-symbols, which correspond to terms or conjunctions of terms in the constraint function. The trick is correctly computing the production probabilities.

3.1 Resolving a simple constraint

```
S: A B | A C | {foo, A, B};
A: i (0.5) | j (0.3) | k (0.2);
B: x (0.6) | y (0.4);

foo {
  i     : x (0.2) | y (0.8);
  j | k : x;
}
```

Figure 7: An SLG grammar with one simple constraint.

Consider the grammar depicted in Figure 7. The source symbol produces either A B or A C, A produces i, j, or k, and B produces x or y. However, when S produces A B, the production out of A should constrain the production out of B. The constraint function, foo, indicates that when A produces an i, B will produce an x with probability (prob.) 0.27273 and a y with prob. 0.72727 (after filtering the base B distribution). But if A produces a j or k, B must produce x.

To resolve the constraint, we start at the source symbol, A. A sub-symbol is created for each term or set of terms that could be matched by a source production. In this case, no productions can match more than one term, but production i matches term 1 and productions j and k match term 2. Therefore, two new symbols are created. A-1S0.1 only produces i, thus matching term 1, and A-1S0.2 produces j or k, thus matching term 2. The relative frequency of j and k should not change, so A-1S0.1 will produce j with prob. 0.6 and k with prob. 0.4. The *source strength* of each of the sub-symbols is the prob. that the original symbol, A, would have produced one of the productions in the sub-symbol. For example, the source strength of A-1S0.1 is the prob. that A produces i, or 0.5. The source strength of A-1S0.2 will be $0.2 + 0.3$, or 0.5 as well.

Now we take a step up towards the root. In this simple case, the source path was only one symbol long so we are now at the root. Each root production that matches the constraint will be split. A production matches this constraint if it contains at least one A and one B, so only the first constraint matches. Production A B will be replaced with a pair of productions, A-1S0.1 B-1G0.1 and A-1S0.2 B-1G0.2. The prob. that A-1S0.1 B-1G0.1 is used is equal to the product of the original prob. of A B, 0.5, and the source strength of sub-symbol A-1S0.1, also 0.5. This is the prob. that the original grammar would have produced an i, which is 0.25. The prob. that A-1S0.2 B.1G0.2 is produced is the product of 0.5 and the source strength of A-1S0.2, which is also 0.25. Note that the overall prob. of producing a j, 0.3, has not changed.

What remains is to define the new B sub-symbols, B-1G0.1 and B-1G0.2. B-1G0.1 is the version of B that should be produced when term 1 of the constraint is satisfied. Thus, the distribution for B-1G0.1 is the base B distribution, $(0.6, 0.4)$, filtered by the first term of foo, $(0.2, 0.8)$, which is be called the constraining distribution. The resulting distribution is $(0.27273, 0.72727)$. B-1G0.2 must satisfy the second term of foo, which has a constraining distribution of $(1.0, 0.0)$, and therefore produces only x. Because symbol B itself is no longer used, the grammar reduction procedure (Section 4) eliminates it. Figure 8 shows the resulting fully-resolved grammar. For such a simple example, the use of the constraint didn't actually help reduce the size of the grammar.

```
S: A-1S0.1 B-1G0.1 (0.25) | A-1S0.2 B-1G0.2 (0.25) | A C (0.5);
A: i (0.5) | j (0.3) | k (0.2);
A-1S0.1: i;
A-1S0.2: j (0.6) | k (0.4);
B-1G0.1: x (0.27273) | y (0.72727);
B-1G0.2: x;
```

Figure 8: The simple grammar with the constraint resolved.

3.2 Resolving a deeper constraint

```
S: A B | A A B | E | {foo, A C, B D};
A: C | C C | E;
C: i | j | k;
B: D | E;
D: x | y;

foo {
  i | j : x (0.2) | y (0.8);
  j : x;}
```

Figure 9: A grammar with a moderately complex constraint.

If we extend the source and goal paths and allow multiple references to a source path symbol in a single production, the process of resolving a constraint becomes more complex. Consider the grammar in Figure 9. The source path and goal path of the constraint now have two symbols each. Additionally, S can produce a production with two A's and A can produce a production with two C's. Finally, the production j from C satisfies two constraints. Each of these factors makes resolving the constraint more complex.

As before, we begin at the source symbol, C. Three sub-symbols will be created. C-1S0 produces only k, which does not match any terms. C-1S0.1 produces i, which matches term 1. C-1S0.1.2 produces j, which matches both terms. All three sub-symbols have a source strength of $1/3$. Now we step up the source path to symbol A. We will have to create six sub-symbols for A to cover all possible combinations of terms that might be satisfied by its productions. The first sub-symbol, A-1S1 should satisfy no terms. Therefore, its productions will be C-1S0, C-1S0 C-1S0, and E. The source strength of symbol A-1S1 is the sum of the *production strengths* for the three productions. A production strength is the original prob. of the production multiplied by the source strengths of any sub-symbols in the production. Thus, the production strength of E is just $1/3$. The production strength of C-1S0 is $1/3 \times 1/3 = 1/9$ and the production strength of C-1S0 C-1S0 will be $1/27$. The total source strength of A-1S1 will therefore be $13/27$. This is the prob. that the symbol A would not have produced an i or a j. The prob. of each production in A-1S1 will be the production strength divided by A-1S1's source strength. In other words, the productions are normalized.

The second sub-symbol produced, A-1S1.1, should lead to productions that satisfy term 1 of the constraint function. Thus, it should always produce a C sub-symbol which itself produces exactly one i. The production E is therefore dropped because it does not contain a C. Production C becomes C-1S0.1 with production strength $1/9$. Production C C is split into two productions, C-1S0.1 C-1S0 and C-1S0 C-1S0.1, each with strength $1/27$. Thus, either A-1S1.1 produces i, i k, or k i. The overall source strength of A-1S1.1 is $4/27$. To take one more example, sub-symbol A-1S1.2x1.2 should satisfy term 1 in two ways and term 2 in one way. Therefore, it must either produce i j or j i. It will have two productions, C-1S0.1 C-1S0.1.2 and C-1S0.1.2 C-1S0.1, each with strength $1/27$.

Now we step up to the root level, S. For each root production that contains at least one A and a B, we will create a sub-production for each combination of terms that we could satisfy by replacing the A's with various A sub-symbols, plus one production in which B is replaced by a sub-symbol that doesn't reach the goal. Production E will remain unchanged, but production A B will be replaced by seven productions.

The first of these is A B-1N1, where B-1N1 is a newly created sub-symbol of B that does not complete the goal path and is therefore not subject to the constraint. In this case, the goal path is B D so B-1N1 cannot produce a D and therefore produces just E. The goal prob. of symbol B is the prob. that B produces a path reaching the goal. In this case, it is the prob. that it produces D, or $1/2$. The prob. of production A B-1N1 will be weighted by one minus the goal prob. and will thus be $1/3 \times 1/2 = 1/6$.

The other six sub-productions will be created by replacing A with one of its six sub-symbols and replacing B with a sub-symbol that is guaranteed to reach the goal and, when it does, produces a goal whose productions have been filtered by the constraining distribution determined by the A's. The prob. of each production will be equal to the product of the original production prob., $1/3$, the prob. that B reaches the goal, $1/2$, and the source strengths of the sub-A symbols used. For example, production A-1S1.2x1.2 B-1G1.2x1.2 has prob. $1/3 \times 1/2 \times 1/27 = 0.00617$. Symbol B-1G1.2x1.2 will be created such that it produces a sub-symbol of D whose distribution is filtered by term 1 twice and term 2 once. However, because term 2 eliminates y, the term 1 filtering has no effect and this symbol just produces x.

The sub-productions for A A B will be even more complex because there are two A's. In addition to the production A A B-1N1 for which B-1N1 doesn't reach the goal, we must form a new production for every way that we can replace the A's by sub-A's. In this case, that will result in $1 + 6 \times 6 = 37$ productions. For each, a sub-B symbol will be created that is guaranteed to reach the goal and is subject to the appropriate term filters.

After having resolved the constraint and reduced the grammar, the resulting SCFG requires 25 non-terminals and 85 productions. Therefore, assuming that the resulting grammar is what we intended, the use of the constraint reduced the original grammar by a factor of about 6.

3.3 Resolving multiple constraints

The constraint resolution process becomes more complicated as we introduce multiple constraints, especially when those constraints share many of the same symbols. Multiple constraints are resolved one at a time. In most cases, the order of resolution does not matter. As we saw in the last section, when a constraint is resolved it generates a number of sub-symbols. When we resolve a second constraint that uses some of those same symbols, the second constraint must be applied to each sub-symbol as it would be applied to the original symbols. As you might well imagine, this has the potential to result in an exponential growth in the number of symbols. Nevertheless, provided that the constraints do not interact too much, fairly large grammars with hundreds of constraints can still be resolved.

The resolution process naturally handles many fairly difficult situations that can arise with multiple constraints. For example, one might wonder what happens when constraints are circular. Consider the grammar in Figure 10. Constraint AB says that if A produces ale then B must produce bed. Constraint BC says that C must then be cat, which, through constraint CA forces A to be awl and so on. A bit of thought should reveal that the only valid sentence in this language is ate big cow, which SLG correctly discovers. If we make the constraints totally circular and add the term cow:ale to function CA, SLG will complain that the start symbol is over-constrained and cannot produce anything.

```
S: A B C | {AB, A, B} | {BC, B, C} | {CA, C, A};
A: ale | awl | ate;
B: bed | bus | big;
C: cat | cry | cow;
AB {ale:bed; awl:bus; ate:big;}
BC {bed:cat; bus:cry; big:cow;}
CA {cat:awl; cry:ate;}
```

Figure 10: A grammar containing circular constraints.

Let us now turn to the problem of resolving two interacting constraints. Consider the grammar in Figure 11. This contains two constraints, {AB, A, B} and {DC, A D, B C}, which we will refer to as constraints 1 and 2, respectively. If constraint 1 is resolved first, we are left with the intermediate grammar shown in Figure 12. We can now resolve constraint 2 as we did in the case of a single constraint, provided we treat the sub-A symbols as A and the sub-B symbols as B. A-1S0.2 does not produce a D, so its production won't be affected. A-1S0.1 only produces D, so we will split it into two sub-symbols, one that produces a sub-D that always produces w and one that produces a sub-D that always produces x. The production A-1S0.1 B-1G0.1 will be split into three sub-productions, each with its own sub-B-1G0.1, as discussed in Section 3.2.

```
S: A B | {AB, A, B} | {DC, A D, B C};
A | B: D | C;
D: w | x;
C: y | z;
AB {D: D (0.2)  | C;
    C: D (0.8)  | C;}
DC {w: y (0.4)  | z;
    x: y (0.6)  | z;}
```

Figure 11: A grammar with two interacting constraints.

```
S: A-1S0.1 B-1G0.1 (0.5) | A-1S0.2 B-1G0.2 (0.5) | {DC, A D, B C};
A-1S0.1: D;
B-1G0.1: D (0.2) | C (0.8);
A-1S0.2: C;
B-1G0.2: D (0.8) | C (0.2);
D: w | x;
C: y | z;
DC {w: y (0.4)  | z;
    x: y (0.6)  | z;}
```

Figure 12: The grammar of Figure 11 after resolving constraint 1.

Figure 13 shows the final grammar, after resolving constraint 1 followed by constraint 2, in a short-hand notation. Each of the four lines represents one production from the root symbol. Numbers preceding colons are probabilities and brackets represent tree depth. For example, the second line indicates that, with 20% prob., a symbol will be produced that produces another symbol that produces a w followed by a symbol that will produce a symbol that produces y 40% of the time and z otherwise.

However, the situation would be more difficult if we had first resolved constraint 2 before constraint 1. Starting with the grammar in Figure 11 and resolving constraint 2 would have left us in the state shown in Figure 14. We can begin as usual by creating new symbols along the source path. However, we will not be able to simply create new goal path symbols whose productions reflect the effect of the constraining distribution because each of the sub-B symbols produces either a C or a D. We will not be able to change the relative frequency of C and D simply by modifying the production probabilities of the goal path symbols. In general, problems like these can occur all along the goal path and can be due to the effects of many previous constraints.

To explain how this situation is resolved, we will have to be more explicit about what really goes on in resolving the goal path and root of a constraint. We begin by defining two terms that relate to the sub-symbols that will be created along the goal path. The *goal distribution* of a symbol B on the goal path is the weighted sum of distributions generated by all goal symbols reachable from B. That is, if we start with B and generate all possible ways of traveling down the goal path to the goal (where we might be using sub-symbols created in resolving previous constraints), the goal distribution will be the average of the production distributions of the goal and sub-goal symbols, weighted by the probabilities of reaching those symbols. When we create a new sub-B symbol that is subject to a certain constraining distribution, the new goal distribution of the sub-symbol should be equivalent to the original goal distribution filtered by the constraining distribution. This is true of all symbols on the goal path. The *goal strength* of the sub-B symbol is the dot-product of the original goal distribution and the constraining distribution.

As before, the process of resolving constraint 1 starts by creating the new source path sub-symbols, and creating sub-productions in the root symbol, S. The set of terms that are satisfied by the sub-A symbols in each of the new root productions determines the constraining distribution for that production. For each sub-production, we will create a new sub-B symbol whose goal distribution has been filtered by the constraining distribution. The new root productions will be as follows:

```
0.5: [C]    [0.8: D | 0.2: C]
0.2: [[w]] [[0.4: y | 0.6: z]]
0.2: [[x]] [[0.6: y | 0.4: z]]
0.1: [D]    [D]
```

Figure 13: The grammar of Figure 11 after resolving both constraints, in a short-hand notation.

```
S: A B-1N1 (0.5) | A-1S1 B-1G1 (0.25) | A-1S1.1 B-1G1.1 (0.125) |
   A-1S1.2 B-1G1.2 (0.125) | {AB, A, B};
A | B: D | C;
B-1N1: D;
A-1S1 | B-1G1: C;
A-1S1.1: D-1S0.1;  D-1S0.1: w;
B-1G1.1: C-1G0.1;  C-1G0.1: y (0.4) | z (0.6);
A-1S1.2: D-1S0.2;  D-1S0.2: x;
B-1G1.2: C-1G0.2;  C-1G0.2: y (0.6) | z (0.4);
AB {D: D (0.2) | C;
    C: D (0.8) | C;}
```

Figure 14: The grammar of Figure 11 after resolving constraint 2.

A-2S0.1	B-1N1-2G0.1
A-2S0.2	B-1N1-2G0.2
A-1S1-2S0.2	B-1G1-2G0.2
A-1S1.1-2S0.1	B-1G1.1-2G0.1
A-1S1.2-2S0.1	B-1G1.2-2G0.1

If the goal path symbol, B, is actually the goal, creating a sub-symbol is easy. We just filter its distribution with the constraining distribution. However, if B is not the goal but is further up on the goal path, producing a constrained sub-symbol is more complex. Let's imagine that the symbol following B on the source path is C. In order to create a sub-B, we first recursively create a sub-C for each B production that uses a C and replace the old C with the constrained one. The prob. of the production is scaled by the goal strength of the sub-C. Once all productions have been scaled, their probabilities are renormalized as follows. First, each group of productions that derived from the same ancestor production in the original grammar is normalized amongst itself so that the sum of probabilities in the group remains the same. If any groups were eliminated, all production probabilities are then normalized across the board.

A similar process occurs in the root symbol. Once the new sub-B has been created, the prob. of the new root production using it is scaled by the goal strength of the sub-B. When this has been done for each new production, the production probabilities are renormalized within groups, where a group is a set of sub-productions that share the same ancestor in the original grammar and which have the same constraining distribution. If any groups died off because they were over-constrained, the productions are normalized overall.

In the case of our example, production A B-1N1, with prob. 0.5 was divided into A-2S0.1 B-1N1-2G0.1 and A-2S0.2 B-1N1-2G0.2. The constraining distribution for the former is (0.2:D 0.8:C) and for the latter is (0.8:D 0.2:C). The initial goal distribution for B-1N1 was (1.0:D 0.0:C). Therefore, the goal strength of B-1N1-2G0.1 was 0.2 and the goal strength of B-1N1-2G0.2 was 0.8. As a result, the final prob. of production A-2S0.1 B-1N1-2G0.1 is 0.1 and the final prob. of production A-2S0.2 B-1N1-2G0.2 is 0.4.

```
0.4: [C]   [D]
0.1: [C]   [C]
0.2: [[w]] [[0.4: y | 0.6: z]]
0.2: [[x]] [[0.6: y | 0.4: z]]
0.1: [D]   [D]
```

Figure 15: The grammar of Figure 11 after resolving constraint 2 followed by constraint 1, in short-hand notation.

The resulting grammar, after both constraints have been resolved, is shown in Figure 15. It is equivalent to the grammar in Figure 13, which was obtained by resolving the constraints in the opposite order, but does not have exactly the same structure. If the first two productions in Figure 15 were combined, they would be equivalent to the first production in Figure 13.

3.4 Constraint conflicts

```
S: A B | {foo, A B, B};
A: C B | {foo, C, B};
B | C: i | j;
foo {i: i (0.99) | j;
     j: i (0.01) | j;}
```

Figure 16: A grammar containing a potential constraint conflict.

```
S: A-1S1.1 B-1G0.1 (0.5) | A-1S1.2 B-1G0.2 (0.5);
A-1S1.1: C B-1S0.1 | {foo, C, B};
B-1S0.1: i;
B-1G0.1: i (0.99) | j (0.01);
A-1S1.2: C B-1S0.2 | {foo, C, B};
B-1S0.2: j;
B-1G0.2: i (0.01) | j (0.99);
```

Figure 17: The grammar of Figure 16 after constraint Y is resolved.

Although most pairs of constraints may be resolved in either order to the same effect, there is one situation in which this is not possible. If the root and goal path of constraint X falls on either the source or goal path of constraint Y, then X must be resolved before Y. The reason is apparent if we consider the example in Figure 16. We will refer to constraint {foo, C, B} as X and to the other constraint as Y. The source of X, A, and its goal path, B, fall on the source path of Y.

If we were to resolve Y first, we would be left with the grammar shown in Figure 17. There are now two sub-A's, each with its own copy of constraint X. But each one produces a sub-B that either produces i or j. We cannot filter the B's goal distributions because they only produce a single symbol. Therefore, it is not possible to resolve X. However, if we were to have resolved X first, it would not have seriously affected the resolution of Y.

Therefore, whenever there is a constraint conflict of this type, the constraints are reordered so the proper constraint is resolved first. However, if the ordering dependencies are circular, there is a problem. SLG gives the user the option of ignoring such conflicts, but a better solution is to restructure the grammar so the constraint ordering is well defined.

4 Minimizing the grammar

The process of resolving the grammar creates many new symbols and productions, some of which may be superfluous. Therefore, after resolution, the grammar is minimized to make it more compact. Because of the tradeoff between the number of symbols and the number of productions, there is no clear definition of a minimal CFG, as there is with a finite state machine. Nevertheless, a number of helpful steps can be taken.

1. **Eliminating epsilon productions**. The first step is to eliminate any epsilon productions from the grammar. This uses a standard algorithm, described in Hopcroft and Ullman (1979), that has been adapted to properly handle the probabilities in a SCFG. It begins by determining, for each symbol, the prob. that the symbol produces only epsilon. This uses an iterative procedure that terminates once the values have adequately settled. Ordinarily this only takes a few iterations, but it could potentially settle rather slowly. It might be possible to formulate a closed-form solution to the epsilon probabilities, but it may involve a system of non-linear equations.

   Once the epsilon probabilities have been determined, for each production that uses one or more symbols with a non-zero prob. of producing epsilon and for each subset of the epsilon-producing symbols in the production, a sub-production is created in which those symbols are eliminated. The prob. of the sub-production is the product of the original production prob., the epsilon probabilities of the symbols that were removed, and the probabilities that the the symbols remaining do not reach epsilon.

2. **Combining Equivalent Productions**. This is a relatively simple step in which any pair of identical productions in a symbol is combined into a single production. Also, any productions with 0.0 prob. are removed.

3. **Removing Unit Productions**. A *unit production* is a production that contains just one non-terminal. That is, one non-terminal is simply replaced by another one. As shown in Hopcroft and Ullman (1979), if a grammar uses unit productions, there is always an equivalent grammar that does not. Because this process can change the structure of the grammar, it is only done when *aggressive* minimization is requested. Although Hopcroft and Ullman (1979) mention an algorithm for removing unit productions in a CFG, it is not efficient for an SCFG.

   The algorithm used in SLG iterates over the non-terminal symbols. For each symbol, A, it first removes any self-unit productions, which are always unnecessary, and renormalizes the remaining productions in A. It then searches in other symbols for any unit productions that use A. These productions are removed and replaced with A's productions, with their probabilities scaled by the prob. of the original production. Any newly created equivalent productions or self-productions are then removed. When this process completes, all unit productions will have been removed.

4. **Removing Equivalent Symbols**. The next step in minimization is to remove any symbols that have identical sets of productions. The reduction process tends to create a lot of these. In order to do this efficiently, the symbols are first sorted based on their productions. Then neighboring symbols are compared and duplicates removed. Any references to the duplicate symbols within productions must be changed to refer instead to the surviving symbol. Unless aggressive minimization is requested, two symbols which were generated from different ancestor symbols during the reduction process are not considered equivalent.

   Because replacing equivalent symbols can create equivalent productions, Step 2 must be repeated. This, in turn, can create more equivalent symbols so Step 4 is run again. This continues until there are no more equivalent symbols. This usually takes just a few iterations.

5. **Removing Unreachable Symbols**. Finally, any symbols that are not reachable from the start symbol, and could therefore not participate in the grammar, are removed.

5 Word prediction

Although it is easy to generate sentences using any form of SCFG, in order to parse sentences and generate next-word likelihood distributions, it is helpful to convert the grammar to a regular form. A number of algorithms have been developed for parsing context-free languages, most notably the CYK algorithm (Hopcroft & Ullman, 1979). Most of these require the grammar to be in Chomsky normal form, in which each production can either consist of a terminal symbol or two non-terminal symbols. Although they are efficient for parsing whole sentences, these algorithms are not well-suited to performing word-prediction given part of a sentence, particularly if we would like to do it iteratively after each word in the sentence.

The method used by SLG relies on a grammar in Greibach normal form, in which each production must begin with a terminal. With the grammar in this form, it is relatively easy to perform word prediction. As the sentence is processed, from left to right, the parser keeps a list of every possible continuation with their associated probabilities. The continuations are in the form of a terminal followed by one or more other symbols. It is therefore easy to generate a distribution of next words. When the next word is processed, continuations not starting with that word are discarded. The first word is dropped from each remaining continuation and, if the new first symbol is a non-terminal, new continuations are created with the first symbol replaced by each of its productions. While, in theory, this algorithm could generate an exponentially large list of continuations for a highly ambiguous grammar, in practice it does quite well on pseudo-natural languages. Natural languages tend to be only mildly ambiguous, especially if semantic constraints are enforced. If there were too much ambiguity, we would not be able to understand them.

5.1 Converting an SCFG to Greibach normal form

In order to convert an SCFG to Greibach normal form (GNF), the algorithm described in Hopcroft and Ullman (1979) was adapted to handle production probabilities. The algorithm need not start with a grammar in Chomsky normal form, but we will relax the restriction that all symbols following the first terminal in a GNF production must be non-terminals. Figure 18 shows a modified version of the first step of the algorithm, indicating how the probabilities of new productions should be calculated to maintain equivalence. Probabilities are listed in parentheses following each production. The notation P_Q refers to the prob. of production Q being generated by its parent symbol.

Once this step is complete, it will be the case that, for all productions, Q, of the form $A_i \rightarrow A_j\gamma$, i will be less than j. Therefore, we can eliminate all such productions by replacing them with all productions formed by replacing A_j with one of its productions, R. The prob. of the new production will be $P_Q \times P_R$. As long as we start with the last symbol and work to the first, we will never introduce a new production that starts with a non-terminal. A similar process can then be performed to replace all productions of the form $B_i \rightarrow A_j\gamma$. Because there can be no $B_i \rightarrow B_j\gamma$ productions, all productions will now begin with a terminal symbol.

6 Discussion

SLG is intended to help users design interesting context-free languages. It is especially useful in creating training environments for machine learning experiments. Without using constraints, it is still a convenient tool for working with stochastic context-free and regular languages, and includes a number of new algorithms for transforming grammars.

for $k \leftarrow 1$ **to** m **do**
  for $j \leftarrow 1$ **to** $k-1$ **do**
    for each production, Q, of the form $A_k \rightarrow A_j\alpha$ **do**
      for each production, R, of the form $A_j \rightarrow \beta$ **do**
        add production $A_k \rightarrow \beta\alpha \;\; (P_Q \times P_R)$
      remove production $A_k \rightarrow A_j\alpha$
  $x \leftarrow 0$
  for each production, Q, of the form $A_k \rightarrow A_k\alpha$ **do**
    $x \leftarrow x + P_Q$
  for each production, Q, of the form $A_k \rightarrow A_k\alpha$ **do**
    add production $B_k \rightarrow \alpha \;\; (P_Q \times (1-x)/x)$
    add production $B_k \rightarrow \alpha B_k \;\; (P_Q)$
    remove production $A_k \rightarrow A_k\alpha$
  for each production, Q, of the form $A_k \rightarrow \beta$, where β doesn't begin with A_k **do**
    add production $A_k \rightarrow \beta B_k \;\; (P_Q \times x/(1-x))$

Figure 18: A modified version of Figure 4.9 of Hopcroft and Ullmann (1979) indicating how to handle production probabilities in converting to GNF.

However, the use of constraints can greatly simplify the writing of pseudo-natural languages by separating the syntax of the underlying grammar from semantic and pragmatic influences and allowing important contingencies to be carefully controlled.

Although the method for specifying SLG constraints is quite powerful, it does have some limitations and there are several possible extensions that may improve it. Currently, all of the constraints for a particular root symbol must be satisfied. Therefore they essentially form a logical conjunction. The grammar could be more flexible if it allowed an arbitrary boolean formula of constraints to be specified for each symbol. For example, one might specify that constraint A and constraint B must apply or constraint C may apply. If we had a language with adjectives and compound nouns, we might wish to produce phrases such as "the happy dog and the sad dog" or "the happy dog and the happy boy", but not "the happy dog and the happy dog", which would be redundant. We could do this by specifying that either the nouns must differ or the adjectives must differ. In the current implementation, this is possible, but much less convenient.

Another shorthand that may be useful is the addition of single-path constraints. That is, one might filter the productions of a goal symbol at the end of a particular path out of the root symbol, but not in a way contingent on context. These would be helpful in simplifying many grammars and could be used to bound the depth of recursion.

SLG is written in C and should compile on most systems. The source for the latest version is available at http://www.cs.cmu.edu/~dr/Projects/SLG/slg.tar.gz

References

Christiansen, M. H., & Chater, N. (in press). Toward a connectionist model of recursion in human linguistic performance. *Cognitive Science*.

Hopcroft, J. E., & Ullman, J. D. (1979). *Introduction to automata theory, languages, and computation.* Reading, MA: Addison-Wesley Publishing.

A SLG Usage

SLG handles two types of files, *grammar files* and *symbol files*. Grammar files typically have a .slg extension and contain an SLG grammar. The sample grammar files described in this report are included with the source distribution in the Examples/ directory.

In order to use a grammar, it must be *resolved*, or converted to standard SCFG form. SLG can then write a symbol file, which stores the symbols and transitions in the grammar in compressed computer-readable form. The symbol file, which normally ends with extension *.sym*, can later be loaded into SLG to avoid repeating the conversion process.

Finally, if a grammar is to be used to make probability predictions, it must be in Greibach Normal Form. Once a grammar has been converted to GNF, it is common to give its symbol file the extension *.gnf* to distinguish it from a *.sym* file containing a grammar that is not in GNF.

usage: slg [commands]

-h	displays this message
-v num	sets the verbosity level. 0 = silent, 3 = maximum, 1 = default
-r num	sets the random number generator seed value
-i	seeds the random number generator based on the time (done automatically at startup)
-d string	sets the symbol separator for sentence output
-c file	loads an SLG file and resolves it to a SCFG
-a	toggles whether cleanup is aggressive. If so, the basic parse tree may be rearranged to compress the grammar. (default: false)
-j	toggles the removal of unused terminals when converting an SLG file (default: true)
-y num	sets the constraint sensitivity. 2, the default, will cause an error if constraints conflict. 1 will produce a warning and 0 will be silent.
-s	prints the grammar to stdout in legible format
-l	lists the terminal symbols in the order in which their probabilities appear (alphabetical)
-o file	saves the current grammar in a binary symbol file
-f file	loads a new grammar from a binary symbol file
-g	converts the grammar to Greibach normal form
-n num	generates sentences using the current grammar
-k num	generates sentences and gives the word predictions
-p file	reads sentences and gives their word predictions
-m num	sets the max number of words per sentence for -n and -k (-1)
-t	toggles whether -n with verbosity ¿= 2 will produce parse trees in long or short format (default: short).
-w	toggles whether -p and -k display the prediction of the first word in each sentence (default: yes)
-e	toggles whether -p and -k display the prediction following the last word in each sentence (default: no)
-x num	calculates the number of parses possible with this grammar that don't go below the specified depth