The general form of a glrgen grammar file is:

```
options
other lines
%%
rules
%%
other lines
```

The `%%` is the separator so that we can tell the different sections from one another.

Options

```
%option [option, option, option ... ]
```

Options are intended to allow you to set things that would otherwise use command line arguments.
*Note: Right now options are ignored and do nothing.*

Other lines - Any other lines at the top or bottom are simply output in the resulting parser file, at the top and bottom of the file respectively.

Rules - Rules fall into two broad categories, grammar rules and tokenizer rules. Grammar rules define the grammar that is to be parsed, while tokenizer rules help define how the input is broken up into tokens and those different tokens are identified. There are many types of rules with varying results, so the rules will be described in separate sections. The general types of rules are:

Grammar Rules

- Grammar Productions - Your basic grammar definition rule. These say what the grammar itself is and provide the semantic action (code) to do something useful when a production is reduced.
- Error Productions - These are much like grammar productions, except they say what to do when something goes wrong.
- Operator Rules - Specify operator precedence and associativity.
- Forbid/Allow Rules - Rules to forbid certain ways of using grammar productions together, these can clarify an easy to understand but ambiguous grammar without forcing you to rewrite the grammar.
- Type Rules - Type rules define the types used by non-terminals in semantic actions.

Tokenizer Rules

- Token fragment rules - These define reusable bits of regular expressions that can be used to construct other regular expressions.
- Ordered tokens rules - These define the tokens that are reachable from a particular state, and how reading specific tokens cause the tokenizer to transition to a different state.

The Grammar Rules

Grammar Productions are your most basic grammar rule, and define the grammar in what is known as BNF notation (The Internet is your friend on this one). Here's an example:

```
assoc_op : "%forbid" "(" [integer_ranges] ")" ;
```

This is actually the definition of the type rule. Nevermind what it means, we can already pick apart some of the particulars by looking at this example. For instance:

- The format of a basic grammar rules is a non-terminal symbol name followed by `":"` followed by some symbols followed by `";"`.
- This rule defines what an `assoc_op` is.
- The terminals `"%forbid"`, `"("` and `")"` are used in the grammar, and you can use string within the production in order to make the rule more readable.
- The symbol `integer_ranges` is also in the grammar somewhere.
- `integer_ranges` is optional because of the use of brackets `[]` in this production.

Also, when giving more than one production for a non-terminal, the time honored `|` may be used to seperate the alternate productions, as such:

```
assoc_op : "%forbid" "(" integer_ranges ")"
         | "%forbid" "(" ")" ;
```

Besides the symbols that define a grammar, a production might also have one or many properties that appear at the end of the rule and are used to define semantic rule, clarify precedence or disambiguate the grammar. Those parts are:

`%label`

Label the production for use with associativity (keyword `%priorities`) rules. More than one production may use the same label.

Example:

```
e : e '+' e %label=addsub ;
e : e '+' e %label=addsub ;
e : e '*' e %label=mul ;
%priorities %mul > %left addsub ;
```

`%action`

Defines the semantic action of a rule, enclosed in `{% %}`.

Example:

```
floating_point_number : integer '.' integer %action {% $$ = tofloat($1,$3); %} ;
```

Within a semantic rule $$, $ with a number and several other tokens have special significance. Take a look at the section *Semantic Actions* for more information.

`%reject`

This will cause a production to be rejected. This is useful if you want to say that a combination of symbols is allowed, except for a few which need to be rejected. Of course you could simply enumerate the valid options as grammar productions, but giving the pattern and then trimming out the unwanted one can also save time and effort.

Example: Floating point numbers are defined here as numbers with a decimal, and that the number before and after the decimal are optional, but a decimal alone is not valid.

```
floating_point_number : [integer] '.' [integer] ;
floating_point_number : '.' %reject ;
```

The first rule is equivalent to

```
floating_point_number : integer '.' integer ;
                      | integer '.'  ;
                      | '.' integer ;
                      | '.'  ;
```

Then the `%reject` rule makes the last variation invalid.

```
%left
%right
%non
%forbid( int -> int, int -> int ... )
```

These are used to decide associativity, or which productions are allowed to appear within other productions. When used as a property, each of these apply the associativity of the production to itself. For a full discussion of associativity see *Production Associativity*.

Example 1: addition is left associative, the production may not nest in itself on the right
```
e : e '+' e %left ;
```

Example 2: this strange ternary op is not allowed to appear within itself in the center non-terminal.
```
e : e '<' e '>' e %forbid(2 -> 2);
```

%forbid allows you to count non-terminals from the right by using negative numbers. If you understand what %forbid is doing, you'll understand that %left, %right and %non are actually just shorthand for `%forbid(2 -> -)`, `%forbid(0 -> -2)` and `%forbid(0 -> -1)` respectively.

```
%rank
```

This will assign a numeric rank to the production. Later, if there is a run-time ambiguity, the rank will be used to decide which of some alternate productions should be preferred over the other.

*Note: The ambiguity checker make use of this, but it is not currently in the GLR parser.*

Example: Given an ambiguous reduction between a declaration and an expression, we want it to be an expression.
```
statement : declaration_statement %rank=1 ;
statement : expression ';' %rank=2 ;
```

```
%short
%long
```

%short and %long are attached to a non-terminal, not at the end of
If %rank fails to resolve an ambiguity, it is sometimes because two productions are valid but one is longer than the other. When you attach these to a non-terminal in a production the choice can be made by choosing the shorter or longer version.

*Note: The ambiguity checker makes use of this, but it is not currently in the GLR parser.*

Example: Choose the longest up front form in an ambiguous parse.

```
word : [%long word] 'A' ;

%resolvable_conflict
```

When you do this you're letting the parser generator know that your reductions are either/or and you will return 0 or 1 from your action in order to indicate the loser and winner respectively. These resolve the conflict by preventing the invalid reduction from appearing in the first place, and the reduce/reduce conflict is reported as resolved.

This is an ambiguous grammar as defined, but in reality is it not because only one or the other production will be applied.

Example: The identifier was declared as either an enum or class earlier.

```
class_or_enum : classname | enumname ;
classnane : identifier %resolvable_conflict
                        %action {% if( ! isclass($1) ) return 0; $$ = $1; %}
enumname : identifier %resolvable_conflict
                        %action {% if( ! isenum($1) ) return 0; $$ = $1; %}
```

Special productions:

Some productions are special, such as the start production and error productions. The start production contains the special symbol <Start>, and error productions contain the special symbol <error>.

Example 1: The start production, the grammar starts with document.

```
<Start> : document ;
```

Example 2: An error production. If an error is encountered the parser will peel the symbols off the stack and try to match against a production that ends in an error, if it finds one the error production is reduced. Parsing may continue unless the action returns 0.

```
document : header first_record <error>
       %action {%
            ob_printf(oberr, "fatal error %s:%d:%d.\n", $file, @@1->$line, @@1->$col);
            return 0;
      %} ;
```

Operator Rules

The operator rules provide a way of giving precedence without relying on parser table tricks specific to the choice of parsing technique. Instead, they provide a way resolve precedence ambiguities using grammar modification only, and thus offer an advantage over the techniques used today.

```
%priorities [modifier] label1 > [modifier] label2 > [modifier] label3 ...
```

The modifiers are `%right`, `%non` and `%forbid`.

Forbid/Allow Rules - This is where glrgen really shines, offering grammar ambiguity cleanup capabilities that is simply unmatched. Rules to forbid or allow specific productions from appearing as

non-terminals in specific placements of other productions. Moreover, these rules allow you to exclude productions from other arbitrarily nested productions, or based on a matching criteria which can actually eliminate an infinite number of ambiguities simultaneously.

```
%forbid production @ production [%from production %from production ...]
%forbid production @ %label=label %op [%from production %from production ...]
%allow production @ production [%from production %from production ...]
%allow production @ %label=label %op [%from production %from production ...]

%op is one of the associativity operator %left, %right, %non or %forbid( int -> int, int
-> int ... ), see the discussion of these operators within grammar rules if you need more
information.

Example 1: An alternate way to do left associativity
e : e '+' e ;
%forbid [e : e '+' e] @ [e : e '+'.e]

Notice the dot in the production after the @ which indicates where the production on the
left is forbidden.

Example 2: An alternate way to do left associativity with labels
e : e '+' e %label=addsub;
e : e '-' e %label=addsub;
%forbid [e : e '+' e] @ %label=addsub %left ;

Example 3: Dealing with if/else dangling-else ambiguity and making use of the %from
clause.

Example 4: A more general case of the dangling-else ambiguity


Both %forbid and %allow are provided so that depending on the task at hand you may give
the rule and the exception and keep the number of rules short.
```

Type Rules define the types used by non-terminals in semantic actions.

```
%type ctype <- non-terminal non-terminal ...

Example: We want the non-terminal triple to retain information inside a class called
Triple.
%type Triple* <- triple ;
triple : a b c %action {% $$ = new Triple($1,$2,$3); %} ;

The Tokenizer rules
```

Token fragment rules - These define reusable bits of regular expressions that can be used to construct other regular expressions.

```
Ordered tokens rules - These define the tokens that are reachable from a particular state,
and how reading specific tokens cause the tokenizer to transition to a different state.
```