

HyGram

Version 3.0

A Notation for Specifying Regular and Context Free Languages

Randall Hyde

Abstract

HyGram is a compact, easy to read, and easy to use notation for specifying context free grammars and syntax directed definitions. It does not require any special (non-ASCII) characters, hence one may easily embed HyGram definitions in programs as comments. Unlike EBNF, Syntax Diagrams, and other notational systems, HyGram inherits and improves upon many of the features from systems like AWK, Flex/Lex, and Yacc/Bison. Therefore, it is more than just "theoretically complete;" it is a practical tool useful for designing and specifying the syntax and semantics of new programming language systems.

1.0 Introduction

There are three standard mechanisms people employ to specify the syntax for a programming language or other system that uses a context-free language: a set of productions using traditional notation, EBNF, and syntax diagrams. Each of these methods has its own set of problems. Either the result is difficult to read and understand (EBNF), the notation requires special symbols that are unavailable in typical character sets (syntax diagrams), or both (traditional CFGs). HyGram is a tool that lets one specify syntax and semantics of a language system. It uses only characters readily available in the ASCII character set. Nevertheless, it is far more readable than standard EBNF notation.

Any student who successfully completes an automata theory course can tell you that the set of regular languages are a proper subset of the context free languages. For any string you generate using a regular expression, there is a corresponding CFG that will also generate that string. Consider the following examples:

Regular Expression:

```
Int_Const:      [ '0'-'9' ]+
```

Context Free Grammar

```
Digit → '0' | '1' | '2' | '3' | '4' | '5' | '6' |  
       '7' | '8' | '9'  
Int_Const → Digit | Digit Int_Const
```

For those who are comfortable with the notation, the Regular Expression (RE) above is simple and easy to understand; the Context Free Grammar (CFG), on the other hand, takes a few seconds to decipher. Clearly, the RE is more easily read than the CFG (by most people, at least).

EBNF is an example of a notation that lets you specify a CFG using a standard character set (e.g., ASCII). Unfortunately, EBNF uses notation similar to the CFG example above (substituting "::=" for "→" since the arrow symbol isn't available in standard character sets).

EBNF uses the following conventions¹:

- Terminal symbols are names or characters that do not contain EBNF *metacharacters*.
- Nonterminal symbols are names enclosed within the "<" and ">" metacharacters.

- A *production* takes the form:

`<nonterminal> ::= right-hand-side`

The *right-hand-side* is a string of grammar symbols (terminals, non-terminals, and metacharacters).

- The "|" metacharacter allows for *alternation*. That is, it lets you combine several productions into one by specifying the left hand side of a production only once, e.g.,

`<nt> ::= A C | A D`

is equivalent to

`<nt> ::= A C`

`<nt> ::= A D`

- The "(" and ")" metasympols allow convenient grouping. The example above can be written using parentheses as:

`<nt> ::= A (C | D)`

- The brackets "[" and "]" specify optional items. For example,

`<nt> ::= A C | A `

is equivalent to

`<nt> ::= A [C]`

- The braces "{" and "}" indicate zero or more repetitions of the items they enclose. The earlier example,

`<Int_Const> ::= <Digit> | <Int_Const> <Digit>`

could be rewritten as

`<Int_Const> ::= <Digit> {<Digit>}`

1. This particular variant is an adaption of the EBNF from "Programming Languages, Structures and Models" by Dershem & Jipping. There are many minor variants of EBNF.

This is a little easier to read than the previous version. Nevertheless, although EBNF is arguably easier to read than standard BNF or a standard CFG, it still leaves a little to be desired. If you need convincing, just take a look at the EBNF for a language like 'C' or Pascal sometime.

The difficulty with reading (and verifying) EBNF grammars for realistic languages has driven many language designer's to adopt the *syntax diagrams* introduced in the *Pascal User's Manual and Report*. These syntax diagrams (also known as *railroad diagrams*) are much more readable and more compact than the corresponding EBNF for the same language. Unfortunately, syntax diagrams use a graphical notation that is impossible to represent with a standard character set. Although modern word processing and page layout programs provide facilities to incorporate graphic images within a document, such facilities are not widely available in text editors so one could not, for example, embed grammar productions within a program's comments².

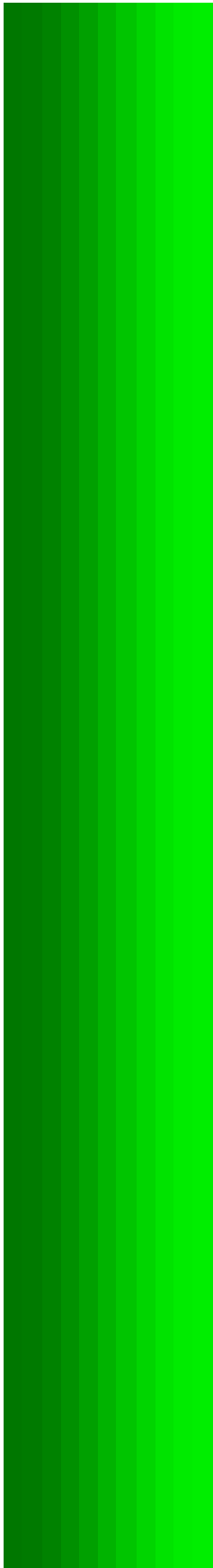
Another problem with syntax diagrams is the amount of space an individual production requires. Overall, syntax diagrams are very compact. However, they achieve this compactness by merging productions in a grammar (the use of arrows in the diagrams eliminates many productions). Unfortunately, the representation of any single production generally takes far more space than an EBNF version of that same production.

One final problem with all the methods mentioned above is that they only specify *syntax*, there is no formal mechanism to specify *semantics* in a system³. Specifying semantics is crucial when formally defining a programming language.

Before marching on and defining HyGram, it is worth pointing out that a real-world partial solution already exists: the YACC (or Bison) programming language. YACC is a "compiler" of context-free grammars⁴. It will compile a CFG in ASCII notation to a push-down automata (PDA). The resulting PDA, in the form of a C program, will recognize strings in that context-free language. In addition, YACC productions let you associate *semantic rules* with each production. The resulting PDA will execute these semantic rules when the PDA *reduces* a right hand side of a production, replacing it by the nonterminal on the left hand side.

Unfortunately, YACC's language is little more than a lexically modified

-
2. Actually, the CodeWright text editor provides a limited facility for this, but this capability is not general nor is it available in many editor systems.
 3. Of course, semantics don't exist in a true context-free system, so this complaint may seem unfair, nevertheless, most programming languages are not context-free and, therefore, need semantics to handle the context dependencies.
 4. Actually, it is limited to the LALR subset of CFGs, but I will ignore that minor point here.



BNF. Although it allows the use of the "|" metacharacter, it does not support the braces, brackets, or parentheses found in EBNF. Most programmers who use YACC (or Bison) tend to use it in conjunction with the LEX (or FLEX) language to process regular expressions separately. This allows the YACC programmer to treat an "integer constant" as a terminal symbol and not bother having to specify the productions to recognize this regular expression. While this makes YACC programs quite a bit more readable than standard EBNF grammars, it does move important components of the grammar (i.e., all the supporting regular expressions) to a different source file that a different language has to process.

Although HyGram must be different than existing methods in order to achieve its goals, it is important that HyGram be sufficiently close to existing notation so a new user will feel comfortable using it. For this reason, HyGram uses many of the notational conventions found in LEX and YACC.

2.0 HyGram

HyGram is a notation that lets a user easily document a *syntax directed definition*. A syntax directed definition is a grammar with a set of associated semantic rules (or *actions*, if you prefer). Formally, a HyGram grammar takes the familiar form:

$$HG = (N, T, P, S)$$

Where N represents a set of non-terminal symbols, T represents a set of terminal symbols, P represents a set of productions (with possible semantic actions), and $S \in N$ is a special *starting production*).

HyGram uses standard (programming language) *identifiers* to denote nonterminal symbols. The following regular expression provides the exact form:

```
{'A'-'Z', 'a'-'z'} {'A'-'Z', 'a'-'z', '0'-'9', '_' }*
```

(Note: braces surround a character set.) That is, a nonterminal begins with an alphabetic character and is followed by zero or more alphanumeric or underscore characters. Note that HyGram does not attach special significance to alphabetic case, so the nonterminal "NT" is equivalent to "nt".

Terminal symbols in HyGram are always strings of one or more characters surrounded by quotes or apostrophes (strings surrounded by apostrophes are exact, those surrounded by quotes are case insensitive). HyGram treats adjacent terminal symbols as a single string, so the following two lines are identical as far as HyGram is concerned:

```
'a' 'b'  
'ab'
```

HyGram uses the backslash character to denote the usual C/C++ style non-graphic symbols like "\n" and "\r". For example, the following string of terminal symbols corresponds to the string "ab" at the end of a line:

```
'ab\n'
```

The starting symbol is a special non-terminal. It is the first production to use when doing a top-down derivation; conversely, it is the symbol you must produce when doing a bottom-up derivation. The very first statement in a HyGram grammar (explained below) describes the starting symbol(s) for a grammar.

Productions in HyGram take the following form:

```
nt (optional_parameters) = rhs  
%  
semantic_actions  
%%
```

A production begins with a nonterminal symbol and ends with the "%%" lexeme. An optional semantic action may follow the right hand

side (RHS) of the production; if present, it begins with a "%" character. The "%" will not be present if the optional semantic rule is not present.

The *nt* item above is an identifier that specifies the nonterminal this production defines (or extends, since a given nonterminal may appear on the left hand side of many different productions). An optional parameter list, containing a sequence of comma separated identifiers, may follow the non-terminal symbol. An equals sign separates the nonterminal (or the parameter list, if present) from the right hand side.

2.1 HyGram Grammar Definitions

A HyGram grammar consists of three components: a single statement that describes the starting symbol, an optional set of terminal symbol definitions, and a set of productions. The first statement of a HyGram grammar typically takes the form:

```
parse nt
```

where "nt" represents a non-terminal symbol that will be the starting symbol for the grammar. The `parse` statement must be the first statement in the source file (i.e., excluding comments and blank lines).

HyGram provides a novel feature that allows you to compose grammars. Specifically, you can specify that the output of one syntax-directed-definition be used as the input to another syntax-directed definition. The first grammar would act as a "filter" for the second. To specify this, one would use the following parse statement:

```
parse nt1 | nt2
```

This informs HyGram to use `nt1` as the actual starting symbol. When the syntax-directed definition first produces some output, that output is fed to the grammar rooted by the starting symbol `nt2`. This forms a "pipeline" of data with the `nt1` grammar filtering out data and feeding the result to the `nt2` grammar. A typical use for this technique is to generate a separate lexical analyzer, parser, and code generator for a typical compiler. One might use a HyGram parse statement like the following:

```
parse lexer | parser | codegen
```

The output of the lexical analyzer (a stream of tokens) becomes the input to the parser. The output of the parser (a parse tree or a stream of codes providing some intermediate representation) is fed to the code generator.

By default, HyGram assumes that the input characters to a grammar (i.e., the set of terminal symbols) are ASCII characters or strings of ASCII characters (emitted from one grammar as input to another). Sometimes, however, it is nice to be able to give generic names to some terminal symbols. You may create a set of names to use for this purpose using the

HyGram **terminals** statement. The terminals statement is optional. However, if it is present it must follow the **parse** statement in the grammar. The terminals statement takes the following form:

```
terminals = (name1, name2, name3, ..., lastname )
```

Names you define with the terminals statement may appear on the RHS of a production wherever a terminal symbol would normally be allowed. You would use the semantic function OUTPUT (described a little later) to insert one of these terminal symbols into a grammar's input stream.

The productions for the grammar(s) must follow the parse statement. HyGram productions take the form mentioned earlier. There must be at least one production for each of the starting symbols appearing in the parse statement.

2.2 Set Definitions and Set Expressions

A very special form of a production is a *set definition* that has the following syntax:

```
{ identifier } = { set_expression } %%
```

This production attaches a name to a character set. The right hand side of this production must be a set definition. A *set_expression* takes one of the following forms:

- A single character surrounded by quotes or apostrophes is a legal set item (quotes vs. apostrophes have the usual meaning).
- A string of characters surrounded by quotes or apostrophes is a legal set item, with each character in the string being a member of the set (quotes vs. apostrophes have the usual meaning).
- The at-sign ("@") is a legal set expression and denotes any legal character.
- A range of items, consisting of a pair of single character constants separated by "->", is a legal set item. Note: both characters must have the same delimiter (quote or apostrophe).
- An identifier, that appears on the left hand side of some other set definition is a legal set item and corresponds to characters in that set.
- If **s** and **t** are legal set expressions, then **s, t** is a legal set item and corresponds to the union of the two sets **s** and **t**.
- If **s** and **t** are legal set expressions, then **s - t** is a legal set item corresponding to all characters in **s** that are not also in **t** (set difference).
- If **s** and **t** are legal set expressions, then **s * t** is a legal set item corresponding to all character that are in both **s** and **t** (set intersection).
- If **X** is a legal set expression, then **(X)** is also a legal set item. It corresponds to the set **X**.
- If **s** is a legal set expression, then **~s** is a legal set corresponding to all

characters that are *not* in *s*.

Note: set union has the lowest precedence followed by set difference, set intersection, set complement, and the set range operator (which has the highest precedence). Union, difference, and intersection are left associative, set complement is right associative, the range operator is non-associative. You may use parentheses to override the precedence.

This particular production form simply defines a set identifier in the HyGram grammar. Therefore, it does not "match" anything. For that reason, you cannot associate a semantic action with this type of production. However, you can easily use the set item in a standard HyGram production and associate a semantic action with that production to achieve the same result.

2.3 HyGram Grammar Components

The right hand side of a HyGram production differs greatly from the standard CFG right hand side. HyGram treats context free grammars as though they were "regular expressions with subroutine calls." Adopting the convention many automata theory books use, the following paragraphs define the right hand side (RHS) recursively. This document will refer to legal RHS components as *grammar components* or GCs. A legal RHS is always a legal GC.

2.3.1 Terminal Symbols

A terminal symbol is the most basic grammar component. All other grammar components, ultimately, are built from terminal symbols. On the RHS of a production, a terminal symbols takes one of three forms: an empty string, an identifier defined within the **terminals** statement, or a literal string constant. Therefore, the following are all legal grammar components (GCs):

- The empty string is a legal GC. Note that a RHS always ends with either the "%" or "%%" lexeme. White space is insignificant. Example of a production that contains an empty string on the RHS:
`Empty = %%`
- If **id** is an identifier defined in the HyGram **terminals** statement, then **id** represents a single terminal symbol and is a legal GC. Note that terminal identifiers are only legal in a grammar if that grammar reads its input from the output of some other HyGram grammar. The **output** function in the semantic action is the only way to output a terminal symbol

for use as input to another grammar.

- A string of zero or more characters surrounded by apostrophies or quotes is a terminal symbol. Strings surrounded by quotes must match an input string exactly. Strings using the quotes delimiters are case-insensitive.

2.3.2 Special Symbols

There are several special symbols that match certain characters, groups of characters, or position on a line of text. Each of these symbols are legal GCs.

- The following special symbols are all valid GCs:
 - ^ Matches the beginning of a line.
 - \$ Matches the end of the line.
 - @ Matches and consumes a single char.
 - ! Matches the end of the file.

The "^", "\$", and "!" are extra special because they do not consume any characters (this is the difference between "\$" and "\n" -- the latter consumes the end of line character).

Special note: The "@" symbol matches any character *except* the end of line character. If you want to match all characters, then use a simple character set containing the items @ and "\n".

These special symbols, of course, have special meaning only outside quotes or apostrophies. That is, "@" matches the at-sign, not an arbitrary symbol.

2.3.3 Non-Terminal Grammar Components

- If **n** is a nonterminal symbol, then **n** is a legal GC and it matches any string that **n** matches⁵. Note that **n** must not require any inherited attributes (that case is handled next).
- If **n** is a legal non-terminal, then so is **n[idlist]**, where **idlist** is a comma separated list of identifiers. This particular GC matches the same strings as **n** (i.e., this GC is effectively equivalent to the one above). It simply attaches a set of names to this particular nonterminal symbol for use in the semantic action section. In the semantic action section, each **id** is equivalent to one of the synthesized or inherited attributes of **n**. HyGram assumes that any particular attribute can be synthesized, inherited, or both (i.e., parameters are in/out parameters). HyGram supports L-attributed

5. "**n**" matches any string that a production with **n** on the left hand side matches.

grammars. This means that all inherited attributes you pass to n must be synthesized attributes of grammar symbols to the left of n in the production or inherited attributes of the production containing n on the right hand side. I.e., if $A = B C n[i] D$ then i may be defined by A , B , or C , but not by D .

2.3.4 Composite Grammar Components

Composite grammar components are GCs that are made up by combining other GCs together. The following describe each of the possible composite GCs.

- If r is a legal terminal symbol, or a string of legal terminal symbols, then r is a legal GC. This GC matches the specified character string. If r is a string literal surrounded by quotes (rather than apostrophes) then the comparison is case insensitive.
- If r is a set item, then r is a legal GC that corresponds to a single character from that set. Examples:

<code>{ 'a' -> 'z' }</code>	Lower case chars.
<code>{ 'A' -> 'Z' , 'a' -> 'z' }</code>	Alphabetics.
<code>{ "A" -> "Z" }</code>	Alphabetics.
<code>{ ' ', \n, \t }</code>	Whitespace.
<code>{ ~ '0' -> '9' }</code>	Non-digit chars.
<code>{ thisSet }</code>	Named set item.
- If r and s are legal GCs, then $r s$ (the concatenation of r followed by s) is also a legal GC. This GC matches the pattern matched by r followed by the pattern matched by s .
- If r and s are legal GCs, then $r | s$ is also a legal GC. This GC matches *either* the pattern matched by r or the pattern matched by s .
- If r is a legal GC, then (r) is also a legal GC. This GC matches the strings that r matches.
- If r and s are legal GCs, then r / s is also a legal GC. This GC matches the same pattern as $r s$ but it does not consume the characters matched by s . This provides a *lexical lookahead* feature.
- If r is a legal GC, then $r?$ is a legal GC that optionally matches the pattern that r matches (i.e., $r?$ matches zero or one occurrences of r).
- If r is a legal GC, then r^* is a legal GC that matches zero or more occurrences of r .
- If r is a legal GC, then $r+$ is a legal GC that matches one or

more occurrences of r).

- If r is a legal GC, then $r\#n$ is also a legal GC. n represents a literal decimal constant. This GC matches n , or more, copies of r .
- If r is a legal GC, then $r\#n\rightarrow m$ is also a legal GC. n and m represent literal decimal constants with $n < m$. This GC matches n through m copies of r .
- If r is a legal GC, then $\langle r \rangle$ is also a legal GC. This GC matches the *shortest sequence* that r will match. This can help eliminate certain ambiguities in a grammar (especially when dealing with regular expressions).
- If r is a legal GC, then $r:id$ is also a legal GC. This associates id with a set of attribute pairs for r . $id.count$ is the number of different strings that r matches (e.g., for the GC $(r:id)^*$ it could turn out that r matches several different strings; $id.count$ tells you how many). $id.lexemes[i]$ is a one-based array⁶ of strings that stores each of the lexemes that r matches; the $id.count$ attribute specifies how many elements are present in this array. If there are attributes associated with r (e.g., $(r[x,y]:id)^*$) then a set of arrays, $id.attrID_1[i] .. id.attrID_k[i]$ (where k is the number of attributes), exists with each array element containing the specified attribute. Example:

$(a[x,y]:s)^*$ has the following attributes:

$s.count$, $s.lexemes[1..s.count]$, $s.x[1..s.count]$, and $s.y[1..s.count]$.

Note that there is a major difference between $(a:s)^*$ and $(a^*):s$. In the first case, $s.count$ is the number of different strings that a happens to match (as just described). In the second example, $s.count$ is always one and $s.lexemes[1]$ is always one the entire string that a matches (although it could be the empty string). E.g.,

$([a'-z'] :s)^*$ matches 'abc' and produces the following:

$s.count = 3$

$s.lexemes[1] = 'a'$, $s.lexemes[2] = 'b'$, and $s.lexemes[3] = 'c'$.

On the other hand, $([a'-z']^*):s$ also matches 'abc' but it produces:

$s.count = 1$ and $s.lexemes[1] = 'abc'$.

6. Brackets attached to attributes have the usual HLL array subscript meaning. Note that these brackets would never appear on the RHS; they would always appear in the semantic action section, so there is no ambiguity.

HyGram allows the abbreviated attribute "lexeme" to represent "lexemes[1]". For example,

s.lexeme

is identical to

s.lexemes[1].

2.4 Operator Precedence and Associativity

HyGram GCs include several operators, including concatenation, '+', '*', '?', '/', '|', '{}', '()', and '<>'. These have the following precedences and associativities:

Table 1: Precedence and Associativity

Item	Prec	Assoc	Desc
ntp[list]	6	left	A non-terminal symbol with a parameter list.
nt	6	left	A non-terminal symbol.
t	6	left	A terminal symbol.
{ }	6	left	Defines a character set.
()	5	left	Groups items inside () to override precedence.
< >	5	left	Groups items inside < > and selects the short possible string to match.
[]	5	left	Defines a parameter list.
#	5	left	Modifies GC immediately to its left to match a specified number of items (or a minimum number of items).
*	5	left	Modifies GC immediately to its left to match zero or more copies of the GC.
+	5	left	Modifies GC immediately to its left to match one or more copies of the GC.
?	5	left	Modifies GC immediately to its left to match zero or one copies of the GC.
:	4	None	Separates an attribute name from a GC.
<i>concat</i>	3	left	If two GCs are adjacent, the resulting GC is the concatenation of them.

Table 1: Precedence and Associativity

Item	Prec	Assoc	Desc
/	2	None	Matches and consumes GC to left of "/" if the characters matched by the GC on the right immediately follow.
	1	left	Matches the GC on the left or the right of this operator.

The "/" symbol is non-associative. One would not normally expect to have two of these adjacent to one another (the second one would be irrelevant). Likewise, the ":" symbol is non-associative, you may have only one symbol attached to a GC and no higher precedence operator may appear immediately to the right of the symbol.

2.5 HyGram Parameters

As pointed out earlier, the left hand side of a HyGram production may contain an optional parameter list. This list contains one or more identifiers separated by commas. Each identifier corresponds to an inherited (input) or synthesized (output) attribute for the production.

```
A[x,y,z] =
    B[x] C[y] D:DsVal
%
    On exit, set y=x and z = DsVal.Lexemes[1].
%%
```

In this example, x (with respect to A) is an inherited attribute, y is both an inherited and a synthesized attribute (since it is passed as input to C and the semantic action above assigns a value to y), and z is a synthesized attribute.

Parameters are *typeless*. Generally, they will be character strings. However, as appropriate you can attach numeric or other values to a parameter if your usage is consistent.

HyGram does not require that all productions for a given nonterminal symbol have the same number of parameters. Indeed, while deriving (or generating) a string using the productions, the parameters are almost irrelevant. The only time a HyGram derivation would use the parameter list is to resolve ambiguity. If one could use two different productions and one production's parameter list matches the parameters associated with the symbol to expand, ambiguity is resolved in favor of the exact match.

Other than the above case, HyGram ignores any extra parameters. If some production refers to those parameter values (attributes), then

HyGram substitutes a NULL value. If the semantic action cannot handle NULL values, then the parse should fail.

2.6 Semantic Rules

The semantic rules section is optional. If it is present, then a single percent sign ("%") separates the right hand side of the production from the semantic action. If the production does not have an associated semantic rule, then the "%" is absent and the right hand side (as well as the entire production) ends with the "%%" lexeme.

With three exceptions, HyGram does not define the contents of the semantic rule section. As far as HyGram is concerned, this is nothing more than a comment. If one were to build a parser generator using HyGram syntax, one would normally embed high level language statements (or even assembly, if you really wanted) into this section. Presumably, the PDA generated by the HyGram parser generator would execute this code sequence whenever it reduces the right hand side of the production to the left hand side.

There are three functions that a HyGram user can assume are available in the semantic action section: Parse, Oput, and Fail.

Parse(nt) will execute the specific grammar specified by the non-terminal passed as a parameter to Parse ("nt" in this case). Parse returns true if the specified sub-grammar successfully matches data arriving at the input stream, it returns false if it cannot match input data. Note that parse consumes terminal symbols from the same input stream as the production whose semantic action calls parse. Warning: note that parse is somewhat dangerous. It allows you to code decidedly non-context free operations into your grammar. I have not studied this feature enough to determine if it will create some inconsistencies in the grammar notation.

Output emits a data output to an output stream. This output stream becomes the input stream of some other grammar if you specified a chain of starting symbols in the original HyGram parse statement. Note that each call to output emits a single terminal symbol for input to the next grammar in the chain. Specifically, note that

```
output "Hello world"
```

emits the non-terminal object "Hello world" it does not emit the sequece of characters "H", "e", "l", "l", "o", " ", "w", "o", "r", "l", "d" to the receiving grammar. If you want to output a stream of characters to the grammar, you must output them one at a time.

You may also emit terminal symbols (defined with the **terminals** statement) to the output stream using the output function. Indeed, this is the only way those terminal symbols would ever appear in the input stream.

You may associate an attribute with a terminal symbol by using an output statements like the following:

```
output IntTerm.Value = 123
output IntTerm.Lexeme = "123"
```

Fail will cause the associated production to reject the current input string.

Note that, unlike YACC and Bison, HyGram does not let you embed semantic rules directly into the middle of a production. There are sound technical reasons why you shouldn't do this, although the primary reason for not allowing it is because it makes the syntax directed definitions harder to read. This is not a restriction on the language, because you can always do the following

```
X = A B ***** C D % Some Rule %%
***** = Some other Rule.
```

Transform this to:

```
X = A B Y C D % Some Rule %%
Y = % Some other Rule. %%
```

Since the Y production matches the empty string, it will always reduce, executing "Some other Rule." at the intended spot.

Of course, what good would a programming language (or notational system, for that matter?) be if you could not insert comments? HyGram uses the semicolon (;) to mark the beginning of a comment. HyGram ignores all characters from a semicolon to the end of the current line. The only exception is within the semantic action where HyGram treats *everything* as a comment. Of course, if you've got a HyGram parser generator, then comments within the semantic rules section are dependent upon the target parser language.

3.0 A HyGram Grammar for HyGram

Since HyGram is, itself, a context free language, it is possible to develop a HyGram grammar for it. This section presents just such a grammar as an example of the HyGram notation.

```
PARSE grammar
```

```
Grammar = ParseStmt
         ( Production | SetDefinition )+
%%
```

```

; White space

ws =
    [' ', \t, \n, \r]*
%%

w =
    (ws | <' ;' @* \n> )*
%%

; Definition of identifiers used by HyGram.

id[Name] =
    (['A'-'Z', 'a'-'z']
     ['A'-'Z', 'a'-'z', '0'-'9', '_' ]*):theID
%
    Set Name equal to theID.Lexeme.
%%

; IdList- Handles one or more comma separated IDs.

IdList[List] =
    id w (',' w id w)*:List
%
    Returns the list of identifiers.
%%

; ParseStmt- Handles the initial statement in
; a HyGram Grammar.

ParseStmt =
    "parse" w
    FilteredID
    ([' ', \t] | (' ;' @* ))* '\n' w %%

FilteredID = id ( w "|" w id )* %%

; Match a terminal string.

```



```

HexEsc = '\\0x' { '0' -> '9', "A" -> "F" } #1 -> 2 %%
Escape = HexEsc |
        '\\n' |
        '\\r' |
        '\\b' |
        '\\t' |
        '\\f' |
        '\\v' |
        '\\a' |
        '\\\' |
        '\"'
%%

```

```

SingleChar =
        { @ - '\ ' } | Escape
%%

```

```

char =
        '"' SingleChar '"' w |
        "'" SingleChar "'" w
%%

```

```

string =
        '"' ( { @ - '"' } | Escape )* '"' w |
        "'" ( { @ - "'" } | Escape )* "'" w
%%

```

; The following production defines a set definition.

```

SetDefinition=
        w '{' w id '}' w '=' w SetExpression
%%

```

```

SetExpression = '{' w Union '}' w '%%'
%%

```

```

Union = Union ',' w Diff | Diff %%
Diff = Diff '-' w Intersect | Intersect %%
Intersect = Intersect '*' w Complm | Complm %%
Complm = '~' w Complm | Range %%
Range = Char '->' w Char | SetItem %%
SetItem =
        '@' w |

```

```

Char |
String|
id |
'(' w Union ')' w
%%

;-----
; The following production defines a HyGram
; production.

Production =
    w id OptParm '=' GC0 OptRule '%%'
%%

OptRule =
    '%' < @* / '%%' >
%%

; Handle the optional parameter section here:

OptParm =
    ( '[' w IdList ']' w )?
%%

; Handle the empty production down here:

GC0 = GC1? &&

; Handle alternation here (lowest precedence).

GC1 =
    ( GC1 '|' w GC2 ) |
    GC2
%%

; Handle lexical lookahead here (precedence level 2).

GC2 =
    ( GC3 '/' w GC3 ) |
    GC3
%%

```

```

; Handle concatenation of GCs here.

GC3 =
    (GC3 GC4) |
    GC4
%%

; Handle the ":" operator for attaching a label
; to a GC:

GC4 =
    GC4 ':' w id
%%

; Handle the */+/?/{}/:/()/<> operators here.
; Also handles individual items.

GC5 =
    GC5 '*' w |
    GC5 '+' w |
    GC5 '?' w |
    GC5 Range |

    SetExpression |

    '(' GC1 ')' w |
    '<' GC1 '>' w |

    '^' w |
    '$' w |
    '@' w |
    '!' w |
    GC6
%%

; Handle terminals and non-terminals down here.

GC6 =
    ntname |

```

```

        char |
        string
    %%

; Handle numeric ranges of GCs.

% IntConst =
    {'0'-'9'}+ w
    %%

Range =
    '#' w IntConst w
    %%

Range =
    '#' w IntConst[low] '->' w
        IntConst[high]
    %
    Fail if atoi(low) > atoi(high).
    %%

; Match non-terminal symbols.

ntName = id[Name]
%
    Name must be a name of a nonterminal
    symbol in the grammar
    %%

ntName = id[Name] '[' w IDList[IDs] '['
%
    "Name" must be the name of a nonterminal
    symbol in the HyGram grammar.
    Each IDs.Lexeme[i] identifier should be
    (otherwise) undefined symbols.
    %%

```