A Grammar Processor

Peter Grogono

Tuesday 29th June, 2010

Contents

Revision History

Date	Revised by	Description
1 Jan 2010	P.G.	`\alt` is built into the code and is not defined in the control file.
1 Jan 2010	P.G.	The new command `%%factor` attempts to left-factorize the grammar.
8 May 2010	P.G.	The program writes a LaTeX macro for each production to the "definitions" file.
29 June 2010	P.G.	Empty symbol added. TBD: document # as empty symbol and other consequences.

1. Introduction

An attractive grammar is desirable for æsthetic reasons and, more importantly, it is much easier to spot errors in a grammar that is cleanly formatted than in a grammar that is a

mess. On the other hand, grammars require a mixture of fonts and are therefore tedious to write in LaTeX. The Grammar Processor translates a grammar written in a simple ASCII format to a formatted LaTeX grammar and also provides additional information that may be of use to parser writers.

2. Usage

The program is called `gramproc` and it is invoked with up to four file names:

```
gramproc input control output definitions
```

This command tells `gramproc` to read from `input.grm` and `control.tex`, to write a report to `output.tex`, and to write definitions to the file `definitions.tex`. The extensions `.grm` and `.tex` are wired into the program and are non-negotiable.

If the fourth argument is omitted, `gramproc` uses the name of the first file with "`-defs`" appended:

`gramproc input control input` ≡ `gramproc input control input input-defs`

If the third argument is omitted, `gramproc` uses the name of the first file, reading from `input.grm` and creating `input.tex`:

`gramproc input control` ≡ `gramproc input control input input-defs`

If the second argument is omitted, `gramproc` looks for a control file called `control.tex`:

`gramproc input` ≡ `gramproc input control input input-defs`

3. The Input File

The input file contains the productions (or *rules*) of the grammar. Figure 1 shows a simple example of a grammar, and Figure 2 shows part of the output generated by `gramdoc` for this grammar.

```
%Here is a \emph{very simple} grammar.
Program = ( Declaration | Assignment ) $ ';' .
Declaration = ?identifier [ ':' Type ] [ ( '=' | ':=' ) Expr ] .
Type = !Char | !Integer .
Assignment = ?identifier ':=' Expr .
Expr = Term $ ('+' | '-') .
Term = Factor $ ('*' | '/') .
Factor = ?identifier | ?number | '(' Expr ')' .
```

Figure 1: A simple grammar

Program = { *Declaration* | *Assignment* }‘;’ .
Declaration = identifier [‘:’ *Type*] [(‘=’ | ‘:=’) *Expr*] .
Type = **Char** | **Integer** .
Assignment = identifier ‘:=’ *Expr* .
Expr = { *Term* }(‘+’ | ‘-’) .
Term = { *Factor* }(‘*’ | ‘/’) .
Factor = identifier | number | ‘(’ *Expr* ‘)’ .

Figure 2: Output generated by `gramdoc` from the grammar of Figure 1

Comments The input file may start with one or more comments. A comment starts with `%` and continues until the next line break. The first line of the grammar in Figure 1 is a comment. Comments may be copied to the output file, as explained in Section 4. They may contain LaTeX commands, provided that the commands are either standard or are defined in the Control file.

All comments must appear before any rules are given.

Metasymbols Certain symbols (the *metasymbols*) play particular roles in the grammar. The metasymbols used by `gramproc` are:

```
=    .    (    )    [    ]    {    }    &    ?    !
```

Each rule has the form

⟨*Symbol*⟩ "=" ⟨*Expression*⟩ "."

in which "=" and "." are metasymbols. The other metasymbols are used to build expressions as follows (α, β, ... denote arbitrary expressions; ϵ denotes the empty string):

- Concatenation corresponds to sequencing.
- A vertical bar denotes an alternative: "$\alpha \mid \beta$" is equivalent to either α or β.
- A dollar symbol, used as an infix operator, denotes a non-empty list in which the right operand is a separator: $\alpha \,\$\, \beta$ is equivalent $\alpha \mid \alpha\,\beta\,\alpha \mid \alpha\,\beta\,\alpha\,\beta\,\alpha \mid \cdots$. For a list that may be empty, use $[\,\alpha \,\$\, \beta\,]$.
- Parentheses are used for grouping: $(\,\alpha\,)$ is equivalent to α.
- Square brackets indicate an option: $[\,\alpha\,]$ is equivalent to $\epsilon \mid \alpha$.
- Braces indicate zero or more repeats: $\{\,\alpha\,\}$ is equivalent to $\epsilon \mid \alpha \mid \alpha\,\alpha \mid \cdots$.

The order of precedence is:

(lowest) | $ concatenation (highest)

The grammar of Figure 1 uses all of these symbols. The first line, for example, says that a ⟨*Program*⟩ is a list of ⟨*Declaration*⟩s and ⟨*Assignment*⟩s separated by semicolons. The use of the list constructor "`$`" in the rules for *Expr* and *Term* is slightly unorthodox, but demonstrates that the list separator may be a general expression, not just a terminal symbol. (These rules also illustrate the precedence rules: the parentheses are necessary.) Note how the list is represented in the output (Figure 2); again, this is unconventional, but avoids the repetition required in the usual format:

Expr = *Term* { (`'+'` | `'-'`) *Term* } .

Nonterminal symbols A nonterminal symbol starts with an upper-case letter and continues with zero or more alphanumeric characters. An *alphanumeric character* is an upper-case letter, a lower-case letter, a decimal digit, or an underscore. More concisely:

⟨*Nonterminal*⟩ `= [A-Z][A-Za-z0-9_]*.`

Terminal symbols Terminal symbols are more complicated than nonterminals. There are five kinds of nonterminal:

- A string consisting of a lower-case letter followed by zero or more alphanumerics is interpreted as a keyword. E.g.: `while`.
- A string consisting of "`?`" followed by one or more alphanumerics is interpreted as an information-carrying symbol. In Figure 1, for example, `?identifier` is a single terminal symbol of the grammar that may have many different realizations in a program.
- A string consisting of "`!`" followed by one or more alphanumerics is interpreted as a keyword. This allows keywords to start with an upper-case letter without being mistaken for a nonterminal symbol. In Figure 1, for example, `!Char` and `!Integer` are treated as keywords.
- A string of the form `"`σ`"` in which σ is a string that does not contain `"`, is a terminal symbol.
- A string of the form `'`σ`'` in which σ is a string that does not contain `'`, is a terminal symbol.

The last two forms allow most symbols to be represented without escape characters. However, it is not possible to define a terminal symbol that contains both `'` and `"`.

There is no special symbol for the empty string, but an expression such as [ ] is equivalent to the empty string. In an EBNF, the empty string is rarely necessary. For example, the expression $\alpha \,|\, \beta \,|\, \epsilon$ can be written more intuitively as $[\,\alpha \,|\, \beta\,]$.

4. The Control File

The control file is a conventional LaTeX source file that `gramproc` copies to the output file. It contains *directives* that LaTeX would treat as comments, if it saw them, but in fact cause `gramproc` to write information about the grammar to the output file. All directives have the following properties:

- Their first two characters are "`%%`" and the following characters are lower-case letters.
- A directive may appear in any position on the line. However, all other text on the line is discarded (not copied to the output file).
- The information generated by a directive has a particular form. The directive must be used in a context that respects that form. The forms are:

  General: one or more LaTeX paragraphs that do not require a particular context.

  List: a sequence of strings separated by `\sep`, a macro explained below.

  Two-column table: a sequence of table rows with structure "A `&` B `\\`". The `tabular` environment must be defined in the control file.

  Three-column table: a sequence of table rows with structure "A `&` B `&` C `\\`". The `tabular` environment must be defined in the control file.

The tables generated by `gramproc` may be very long. For example, the grammar for a production-quality language may have several hundred productions. It is therefore best to use `supertabular` environments rather than `tabular` environments, so that tables can run over page boundaries.

The directives are explained next. With a few exceptions, noted below, directives may be given in any order and are optional. For example, if you do not wish to see errors, do not include the directive `%%errors`.

`%%filename` (General): the name of the file containing the gramamr.

`%%comments` (General): copy any comments that appeared at the start of the input file. Each comment is set as a separate paragraph.

`%%errors` (General): a report of errors detected while processing the grammar. `gramproc` will abort if there are serious errors. Currently, `gramdoc` reports nonterminal symbols that are:

1. defined more than once;
2. not defined;
3. defined but not used in any productions.

`%%factor` (General): `gramproc` attempts to left-factorize the grammar. For example, the result of factorizing the grammar

```
A = B | C .
B = x Btail .
C = x CTail .
```

would be

```
A = x ( Btail | Ctail ) .
```

The directive `%%factor` performs the factorization and writes a list of rules removed by factoring. In the example above, rules `B` and `C` would be removed. To see the factorized grammar, use the directive `%%rules` described below.

`%%statistics` (two-column table): generates a short table giving the values of various counters.

`%%rules` (three-column table): the rules of the grammar. Each rule has the form

⟨*Symbol*⟩ `&=&` ⟨*Expression*⟩ `\\`

in which ⟨*Expression*⟩ may be very long. If the expression has the form $\alpha_1|\cdots|\alpha_n$, it is displayed with one line for each alternative.

`%%sortalpha`: generates no output, but specifies that rules should be output in alphabetical order of nonterminals. This directive affects only instances of `%%rules` that follow it in the control file.

`%%sortinput`: generates no output, but specifies that rules should be output in the order in which they appear in the input file. This directive affects only instances of `%%rules` that follow it in the control file.

By default, the rules are sorted according to their order of appearance in the input file, as specified by `%%sortinput`.

`%%nonterminals` (List): the nonterminal symbols of the grammar.

`%%terminals` (List): the terminal symbols of the grammar.

`%%nullables` (List): the nullable nonterminals — that is, the nonterminals that may produce the empty string.

`%%startsymbols` (two-column table): the start symbol set for each nonterminal. This set contains the terminal symbols that may be produced as the first symbol of the nonterminal. Each start set has the form

⟨*Symbol*⟩`:&` ⟨*List*⟩ `\\`

in which ⟨*List*⟩ may be very long.

`%%followsymbols` (two-column table): the follow symbol set for each nonterminal. This set contains the terminal symbols that may follow a string prduced by the nonterminal. Each start set has the form

⟨*Symbol*⟩`:&` ⟨*List*⟩ `\\`

in which ⟨*List*⟩ may be very long.

`%%overlaps` (three-coumn table): situations in which a strict LL(1) parser would not know how to proceed. The three columns of the table contain: the rule name; the overlapping terminal symbol; and the expressions that overlap.

For example, the grammar of Figure 1 produces the overlap report shown in Figure 3. This report says that the rule for *Program* contains two expressions that are alternatives (*Declaration* and *Assignment*) that both start with the terminal identifier.

Rule	Symbol	Expressions
Program	identifier	*Declaration*
		Assignment

Figure 3: An overlap report for the grammar of Figure 1

`%%factor` (general): factorize the grammar in order to remove overlaps. No rules are added, but some rules may be subsumed by others. For example, when the grammar of Figure 1 is factorized, the rule for *Program* becomes

Program = identifier ([`':'` *Type*] [*AssignOp Expr*] | *AssignOp Expr*) .

and rules *Assignment* and *Declaration* are removed from the grammar. The factorized grammar may be harder to understand than the original grammar, but it may be of more use to a parser writer.

In addition to the directives, the control file must define several LaTeX commands. The required commands, and suitable definitions, follow.

`\newcommand{\nonterminal}[1]{\mbox{\textsl{#1}}}`
Specifies that nonterminal symbols will be set in a slanted font without hyphenation.

`\newcommand{\keyword}[1]{\mbox{\textbf{#1}}}`
Specifies that keywords will be set in a bold font without hyphenation.

`\newcommand{\terminal}[1]{\mbox{\texttt{#1}}}`
Specifies that terminal symbols will be set in a typewriter font without hyphenation. Typewriter font is strongly recommended for terminals because some characters (e.g., "<" and ">") produce strange results with other fonts.

`\newcommand{\infoterminal}[1]{\mbox{\textsf{#1}}}`
Specifies that information-carrying terminal symbols will be set in a sans-serif font without hyphenation.

`\newcommand{\sep}{\hskip 1em plus 3em}`
Specifies that list items will be separated by very elastic glue. This allows LaTeX to justify long lists without hyphenation even when the list items themselves are long. (The result is a bit ugly, but the alternative — using less elastic glue — tends to give a plethora of overfull hboxes.)

The command `\alt` is no longer required; `gramproc` writes "␣|␣" (LaTeX " `\textbar\` ") to the output file.

5. The Report File

The output file will contain any text provided in the control file, comments copied from the input file, and the requested information about the grammar.

As mentioned in the introduction, the main purpose of the output file is to provide a readable version of the grammar. The current version of `gramproc` does a fairly good

job, but could be improved. The most obvious deficiency is poor line breaks — a human writer could do much better.

Parentheses in the input file are used to control parsing; they are not stored internally. Parentheses in the output file are inserted in accordance with precedence rules. Consequently, it is a good idea to read the formatted grammar carefully to ensure that its structure is as intended. Forgetting that concatenation binds more tightly than the list operator `$`, for example, you might write

```
ParameterList = '(' Expr $ ',' ')' .
```

in your grammar, only to be surprised by the output:

ParameterList = { '(' *Expr* }$_{(\ ',' \ ')'\ )}$.

The choice of quote symbols from LaTeX fonts is difficult. The following choices seem to constitute an acceptable compromise for terminal symbols. Example strings are set in a large font to make distinctions clear.

- Terminal symbols are enclosed in left and right apostrophes (i.e., *grave* and *acute* accents) from the typewriter font. E.g., `=` .
- When the terminal symbol itself is a left or right apostrophe, a sans-serif font is used. E.g., `‘` and `’` .
- When the terminal symbol itself is a quote, typewriter font is used. E.g., `"` .
  The symmetric symbol is appropriate because ASCII does not distinguish `“` and `”` .

6. The Definitions File

The definitions file, which is created unless the grammar cannot be parsed, contains one LaTeX macro definition for each production of the grammar. Here is a typical macro definition (with line breaks inserted):

```
\newcommand{\Assignment}{\nonterminal{Assignment}  & = &
\infoterminal{identifier}  \nonterminal{AssignOp}  \nonterminal{Expr}
. \\ }
```

The macro name is the same as the production name. This might cause a problem if a macro with the same name has already been defined. However, conflicts are unlikely because most standard LaTeX macros begin with a lower case letter.

The format of the rule assumes a three-column table. Here is a typical use of the macro just defined, with the formatted result shown on the right:

```
\begin{tabular}{rcl}
\Assignment
\end{tabular}
```

Assignment = identifier *AssignOp Expr* .

A. Algorithms

The algorithms for computing nullable symbols, start sets, etc., given in compiler text books usually assume a simple form of grammar rules, such as basic BNF. The algorithms for calculations with more general grammar rules, often called EBNF, are fairly straightforward extensions of the standard algorithms, but nevertheless are of sufficient interest to record explicitly. That is the purpose of this section.[1]

A.1. Notation and Conventions

Each algorithm is a fixed-point calculation, using the idea that $\lim_{n\to\infty} f^n(\bot)$ is a fixed-point of f for some suitable $\bot$. For example, if we are computing a set, $\bot$ would be the empty set and applying f might yield a larger set. Since the universe is finite (e.g., the terminal symbols of the grammar), eventually we find $f^{n+1}(\bot) = f^n(\bot)$ and we can stop.

In this section, fixed point calculations are written as

initialization
`repeat`
    `for each` $N = \alpha$ `do:`
        ...
`until` *no changes*

in which $N = \alpha$ denotes a rule of the grammar.

The fixed point computation mentions some function, f say. It is followed by a list of rules that define f for each of the arguments shown in Figure 4.

An implementation must handle the "*no changes*" condition correctly. For example, adding an element x to a set S is a "change" only if S did not contain x already. Since this action (adding an element to a set) is used several times, we use an abbreviation analogous to `+=` for add and assign:

$$S \cup\!= x \qquad \equiv \qquad S := S \cup x.$$

T :	a terminal symbol
N :	a nonterminal symbol
α :	an arbitrary syntactic expression (similarly, β, etc.)
$\alpha_1 \ldots \alpha_n$:	a sequence: α_1 followed by α_2, etc.
$[\,\alpha\,]$:	an optional α
$\{\,\alpha\,\}$:	a repeated α
$\alpha_1 \vert \cdots \vert \alpha_n$:	choice: one of α_i, $1 \le i \le n$
$\{\,\alpha\,\}_\beta$:	a list of α's separated by β's ($\alpha\,\$\,\beta$ in grammars)

Figure 4: Argument forms for syntax functions

[1] Another, no less important, purpose, is to ensure that the implementor can figure them out when revisiting the code a few years from now.

A.2. Nullable

The function *nullable* returns a **Boolean** such that *nullable* α is **true** if α can produce the empty string.

The set *nullset* contains the nullable nonterminals.

Fixed point calculation:

$nullset := \emptyset$
`repeat`
    `for each` $N = \alpha$ `do`:
        `if` *nullable* α `then` $nullset \cup= \{ N \}$
`until` *no changes*

Rules:

$$\begin{aligned} nullable\ T &= \textbf{false} \\ nullable\ N &= N \notin nullset \\ nullable\ \alpha_1 \ldots \alpha_n &= \forall i \,.\, nullable\ \alpha_i \\ nullable\ [\,\alpha\,] &= \textbf{true} \\ nullable\ \{\,\alpha\,\} &= \textbf{true} \\ nullable\ \alpha_1 | \cdots | \alpha_n &= \exists i \,.\, nullable\ \alpha_i \\ nullable\ \{\,\alpha\,\}_\beta &= nullable\ \alpha \end{aligned}$$

A.3. Start sets

The function *start* returns a set of terminal symbols. *start* α contains the terminal symbols that can start a string produced by α.

The map $startsets(\cdot)$ maps nonterminals to sets of terminal symbols. The set $startsets(N)$ is the start symbol set for nonterminal N.

Fixed point calculation:

`for each` N `do`:
    $startsets(N) := \emptyset$
`repeat`
    `for each` $N = \alpha$ `do`:
        $startsets(N) \cup=$ *start* α
`until` *no changes*

Rules:

$$
\begin{array}{rcl}
\mathit{start}\ T & = & \{\, T \,\} \\
\mathit{start}\ N & = & \mathit{startsets}(N) \\
\mathit{start}\ \alpha_1 \cdots \alpha_n & = & \texttt{if}\ \mathit{nullable}\ \alpha_1 \\
 & & \quad \texttt{then}\ \mathit{start}\ \alpha_1 \cup \mathit{start}\ \alpha_2 \cdots \alpha_n \\
 & & \quad \texttt{else}\ \mathit{start}\ \alpha_1 \\
\mathit{start}\ [\, \alpha \,] & = & \mathit{start}\ \alpha \\
\mathit{start}\ \{\, \alpha \,\} & = & \mathit{start}\ \alpha \\
\mathit{start}\ \alpha_1 | \cdots | \alpha_n & = & \bigcup_i \mathit{start}\ \alpha_i \\
\mathit{start}\ \{\, \alpha \,\}_\beta & = & \mathit{start}\ \alpha
\end{array}
$$

A.4. Follow sets

The procedure $\mathit{follow}(\cdot, \cdot)$ constructs a follow set for each nonterminal. $\mathit{follow}(\alpha, S)$ processes the case in which the expression α may be followed by any terminal in the set S.

The map $\mathit{followsets}(\cdot)$ maps nonterminals to sets of terminal symbols. The set $\mathit{followsets}(N)$ is the follow symbol set for nonterminal N.

Fixed point calculation:

```
for each N do:
    followsets(N) := ∅
repeat
    for each N = α do:
        follow(α, followsets(N))
until no changes
```

Rules:

$$
\begin{array}{rcl}
\mathit{follow}(T, S) & = & \text{skip} \\
\mathit{follow}(N, S) & = & \mathit{followsets}(N) \cup\!= S \\
\mathit{follow}(\alpha_1 \ldots \alpha_n, S) & = & \mathit{follrec}(\alpha_1 \ldots \alpha_n, S) \\
\mathit{follow}([\, \alpha \,], S) & = & \mathit{follow}(\alpha, S) \\
\mathit{follow}(\{\, \alpha \,\}, S) & = & \mathit{follow}(\alpha, S \cup \mathit{start}\ \alpha) \\
\mathit{follow}(\alpha_1 | \cdots | \alpha_n, S) & = & \forall i \,.\, \mathit{follow}(\alpha_i, S) \\
\mathit{follow}(\{\, \alpha \,\}_\beta, S) & = & \mathit{follow}(\alpha, S \cup \mathit{start}\ \beta);\ \mathit{follow}(\beta, \mathit{start}\ \alpha)
\end{array}
$$

Follow sets for a sequence $\alpha_1 \ldots \alpha_n$ are computed by calling $\mathit{follrec}(\alpha_1 \ldots \alpha_n, S)$. Clearly, for each pair $\alpha_i \alpha_{i+1}$, the start set of α_{i+1} must be added to the follow set of α_i. Also, S must be added to the follow set of α_n.

However, we must also take care of the case $\alpha_i \alpha_{i+1} \alpha_{i+2}$ in which α_{i+1} is nullable by adding the start symbols of α_{i+2} to the follow set of α_i, as shown here:

```
follrec(α_1 ... α_n, S):
    follow(α_n, S)
    if n > 1:
        if nullable α_n
            then follrec(α_1 ... α_{n-1}, S ∪ start α_n)
            else follrec(α_1 ... α_{n-1}, S)
```

A.5. Overlaps

Overlap calculation differs from the others in that it can be computed directly in one pass, and does not require a fixed-point iteration. The overlap procedure searches each rule of the grammar generating tuples (N, α_i, α_j, S) with the following properties:

- There is a rule $N = \alpha$ in which α contains a subexpression $\alpha_1|\cdots|\alpha_n$.
- There exist i and j such that $1 \leq i < j \leq n$, $S = \mathit{start}\ \alpha_i \cap \mathit{start}\ \alpha_j$, and $S \neq \emptyset$.

This procedure produces too many overlap reports. The number could be reduced by generating tuples (N, E, S) with these properties:

- There is a rule $N = \alpha$ in which α contains a subexpression $\alpha_1|\cdots|\alpha_n$.
- E is a subset of $\{\alpha_1, \ldots, \alpha_n\}$ with at least two elements.
- For every pair $\alpha_i \in E$ and $\alpha_j \in E$: $S = \mathit{start}\ \alpha_i \cap \mathit{start}\ \alpha_j \neq \emptyset$.