# `gramex` - a tool for extracting grammar rules from typeset documents

Adrian Johnstone

February 23, 2012

Department of Computer Science
Egham, Surrey TW20 0EX, England

## Abstract

In the C language community there is a tradition of using a minimalist grammar specification format that is easy for humans to read, but not directly suitable for input to common parser generators. `gramex` is a tool that reads plain-ASCII versions of these grammar specifications and outputs a more familiar Bison-like version. `gramex` has been used to extract grammars from the Kernighan and Ritchie ANSI-C book, the ANSI C++ standard and the Java Language Specification; and a special *Pascal* mode can be used to extract rules from the Pascal standards documents. An accompanying tool `gramconv` can be used to convert `gramex` generated files into other formats, and to perform certain grammar translations such as converting extended BNF into plain BNF.

`gramex` been built, run and tested using the Free Software Foundation's GNU `g++` compiler version 3.3.3 running under Cygwin on Windows-XP.

## Typographical conventions

We use `this font` to represent literal input fragments and ***`this font`*** to represent input fragments which should be substituted with an application specific literal text.

# 1 Introduction

In the C language community there is a tradition of using a minimalist grammar specification format that is easy for humans to read, but not directly suitable for input to common parser generators. In standards documents and some text books, different fonts are used to distinguish terminals and nonterminals, but this formatting information is lost in plain-ASCII versions of the documents. In addition, plain-English comments are sometimes used to specify hard-to-capture constraints on valid character and terminal sequences.

`gramex` is a tool that reads plain-ASCII versions of these grammar specifications and outputs a Bison-like [**?**] version. `gramex` has been used to extract grammars from the Kernighan and Ritchie ANSI-C book, the ANSI C++ standard and the Java Language Specification. An accompanying tool `gramconv` [**?**] can be used to convert `gramex` generated files into other formats, and to perform certain grammar translations such converting extended BNF into plain BNF.

# 2 Downloading and using `gramex`

The `gramex` source code is a single file `gramex.c` which may be downloaded from the RHUL Compiler Group's website under `http://www.cs.rhul.ac.uk`.

`gramex` is written in ANSI-C and compiles under the Free Software Foundation's GNU `g++` compiler using

```
g++ -ansi -pedantic gramex.c
```

Input to `gramex` should be an ASCII file containing the plain text version of the rules. Run the tool with a command like:

`gramex` *`options sourcefile`*

where *`sourcefile`* is the name of the plain ASCII input, and *`options`* may be zero or more of:

- `-c` suppress comment (non-rule) lines
- `-e` treat [ ] and {} as EBNF meta symbols
- `-i` treat productions indented by more than four spaces
- `-p` process Pascal standard file (implies -e) as continuations

The translated output is sent to the console: use the output redirection operator (`>`) to capture it to a file:

```
gramex -c myfile.raw > myfile.gex
```

# 3 Rule extraction

Here is a fragment from the grammar included in the ANSI C++ standard.

*exponent-part:*
`e` *sign*$_{opt}$ `digit-sequence`
`E` *sign*$_{opt}$ `digit-sequence`

*sign:* one of
`+` `-`

In this grammar format:

- ◇ grammar elements are delimited by whitespace;
- ◇ grammar elements representing terminals are written in a `teletype` font;
- ◇ grammar elements representing nonterminals are written in an *italic* font;
- ◇ the first line of a new grammar rule contains a nonterminal name concatenated with a colon *:* as its first element, which may optionally be followed by the elements one of (no other elements may appear on the line);
- ◇ the ordinary rules, each successive line is a production;
- ◇ for rules using the one of construction, each element is a production;
- ◇ a rule is terminated by either a blank line or the start of a new rule;
- ◇ elements may be suffixed by *opt* in which case they are optional; the production containing the optional element is intended to be duplicated once with the optional element and once without.

It is not hard to manually construct source files for, say, YACC from these kinds of typeset grammars, but in practice the process is rather error prone as humans are not very good at accurately transcribing punctuation-like characters. Recently produced standards are usually available in electronic form as either Adobe Acrobat or HTML files, and we can use the appropriate viewers to select and copy grammar rules into a conventional text editor. The resulting ASCII files retain character values and indentation, but suppress font information. Here is the plain-ASCII version of the above extract.

```
exponent-part:
    e signopt digit-sequence
    E signopt digit-sequence

sign: one of
    +   -
```

`gramex` converts such fragments into a Bison-like format in which

- ◇ terminals are stropped with `"` characters,
- ◇ nonterminals are written with underscore (`_`) characters instead of hyphens,
- ◇ productions are separated by vertical bars `|` and terminated with semicolons, and
- ◇ optional parts are suffixed with the `?` optional regular operator. (Bison does not support regular expressions, but the accompanying tool `gramconv` may be used to translate regular expressions into plain BNF rules.)

The `gramex` output for this example is:

```
exponent_part:
  "e" sign? digit_sequence |
  "E" sign? digit_sequence ;

sign:
  "+" | "-" ;
```

# 4 Other text

Grammar rules are often embedded in other text, and some standards use plain-English commands with the grammar to specify tricky constructs.

Within rules, plain-English embedded commands are treated as productions which will generate unhelpful results that must be manually corrected.

Lines which `gramex` does not think are part of valid grammar rules are copied to the output as comment lines prefixed by a double slash `//`.

Occasionally, `gramex` will encounter a comment line that has the general form of a rule start. The C++ standard contains several instances of text paragraphs that begin 'Thus:', for instance. These lines are highlighted by being prefixed with `//??` It is important to review the whole output of a `gramex` run before submitting it to further processing: lines preceded with `//??` are particularly likely to need attention.

The `-c` option causes `gramex` to suppress all comment and blank lines from the output leaving a single blank line between the rules.

# 5 Extensions for the Java Language Specification

The Java Language Specification documents include two grammars: a 'pedagogic' grammar sprinkled throughout the text which is intended to support a topic-led description of the language and the 'development' grammar summarised in a late chapter which is intended to form the basis of development tools.

The development grammar uses two extensions to the basic scheme employed in the ANSI C and ANSI C++ standards documents.

1. The [ ] and {} brackets stand for the optional and Kleene-closure (zero-or-many) operations.

   The `-e` option forces `gramex` to treat these `[`, `]`, `{` and `}` characters as EBNF meta-characters instead of tokens. This of course leaves the problem of how to recognise literal brackets and braces in the grammar. The work-around is to preface these literals with some unusual prefix such as `@!@`. `gramex` will treat `@!@[` as just another terminal, outputting `"@!@["`. A global search and replace may then be used in your favourite editor to convert the `"@!@["` tokens to `"["`.

2. Long productions are split over multiple lines by indenting the continuation lines.

   The `-i` options causes `gramex` to treat productions that are indented by more than four characters as continuation lines.

# 6 Extracting rules from Pascal standards

# 7 Limitations

Since `gramex` is dealing with plain-ASCII grammar rules which have been stripped of their formatting, some unavoidable ambiguities arise. In this section we have tried to list known problems which should be manually checked for.

1. When processing C-style standards `gramex` decides whether an alphanumeric grammar element is a nonterminal or a terminal by looking at its list of valid left hand sides. In the ANSI C++ grammar, for instance, there is a rule for nonterminal *operator:*, but `operator` is also a keyword of C so grammar element operator will always be interpreted as a nonterminal. For Pascal standards, the stropping conventions allow terminal and nonterminals to be directly recognised, so this problem does not arise.

2. When extracting grammars using the `-e` option, instances of `[`, `]`, `{` and `}` elements will always be treated as EBNF meta-characters.

3. When extracting grammars using the `-i` option, productions that are indented more than four spaces will be treated as continuation lines.

4. When extracting grammars using the `-i` option, a maximum of one continuation line is allowed per production, that is, no production may span more than two lines.

# 8 Implementation

`gramex` is written in ANSI-C and may be compiled with standard C and C++ compilers. The main data structures are the input `buffer` (declared line 26 and initialised in lines 215–249) and the `lines` array which is declared on lines 28–34 and created at line 256.

The basic approach is to load the entire input into `buffer`, count the number of lines in `linecount` and then create `lines`, initialising the `start` fields to point to the first character in each line.

Each line is annotated with a *kind*, such as blank, comment, rule start and so on. Kinds are represented by elements of the `enum` declared in lines 23–24.

Line kinds are computed in lines 271–350 in a series of passes. The output is produced by walking the lines array and outputting each line of input under the control of a `switch` statement that tests the line kind at lines 358–416.

```
1   /*****************************************************************************
2   *
3   * gramex version 2.2 by Adrian Johnstone (A.Johnstone@rhul.ac.uk)
4   *
5   * Created 21 August 2006.
6   * V2.0 added Pascal functionality 12 January 2007
7   * V2.1 added indentation for Pascal 19 January 2007
8   * V2.2 added support for ... metasymbol in the C# standards
9   *
10  * gramex.c - extract grammar rules from plain text version of standards.
11  *
12  * A conversion tool that reads raw text versions of the ANSI C,
13  * ANSI C++, Java and Pascal standards and extracts EBNF rules.
14  *
15  * The output may be further processed by gramconv to give rules files in
16  * various formats, and to convert between BNF and EBNF.
17  *
18  * This file may be freely distributed.
19  *
20  * Please mail improvements to the author.
21  *
22  *****************************************************************************/
23  #include<stdio.h>
24  #include<string.h>
25  #include<stdlib.h>
26  #include<ctype.h>
27
28  enum {K_COMMENT, K_MAYBE_TYPO, K_PASCAL_START_RULE, K_PASCAL_START_END_RULE,
29        K_PASCAL_CONTINUATION, K_PASCAL_END_RULE, K_START_RULE,
30        K_START_RULE_ONE_OF, K_END_RULE, K_PRODUCTION, K_PRODUCTION_CONTINUED,
31        K_SECTION_HEADING, K_BLANK};
32
33  char *buffer;
34  unsigned linecount = 0;
35  struct line_struct {
36    char *start;
37    char *lhs_start;
38    char *lhs_end;
39    int kind;
40    char* lhs_nonterminal;
41  } *lines;
42
43  int suppress_comments = 0;
44  int use_dots = 0;
45  int use_ebnf = 0;
46  int use_indentation_as_continuation = 0;
47  int pascal_mode = 0;
```

```
48  int suppress_section_numbers = 0;
49  char *scan_start, *scan_end;
50  int first_production_indent;
51
52  void scan(void)
53  {
54    while (*scan_end != '\n' && isspace(*scan_end))
55      scan_end++;
56
57    scan_start = scan_end;
58
59    while (!isspace(*scan_end))
60      scan_end++;
61  }
62
63  int rest_of_line_empty(void)
64  {
65    char *temp = scan_end;
66    int return_value = 1;
67
68    while (*temp != '\n')
69    {
70      if (!isspace(*temp))
71        return_value = 0;
72      temp++;
73    }
74
75    return return_value;
76  }
77
78  int has_opt_suffix(void)
79  {
80    return *(scan_end - 1) == 't' &&
81           *(scan_end - 2) == 'p' &&
82           *(scan_end - 3) == 'o';
83  }
84
85  int initial_alpha_has_colon_suffix(void)
86  {
87    return isalpha(*scan_start) && *(scan_end - 1) == ':';
88  }
89
90  int is_section_number(void)
91  {
92    int return_value = 1;
93
94    char * temp;
95
96    for (temp = scan_start; temp < scan_end; temp++)
97      if (!(isdigit(*temp) || *temp == '.'))
98        return_value = 0;
99
100   return return_value;
101 }
102
103 int is_valid_pascal_nonterminal(void)
104 {
```

```
105     int return_value = 1;
106
107     char * temp;
108
109     for (temp = scan_start; temp < scan_end; temp++)
110       if (!(isalpha(*temp) || *temp == '_'))
111         return_value = 0;
112
113     return return_value;
114   }
115
116
117   int line_ends_with_period(int line_number)
118   {
119     char *temp = lines[line_number + 1].start - 1;
120
121     while (isspace(*temp))
122       temp--;
123
124     return *temp == '.';
125   }
126
127   int line_ends_with_equals(int line_number)
128   {
129     char *temp = lines[line_number + 1].start - 1;
130
131     while (isspace(*temp))
132       temp--;
133
134     return *temp == '=';
135   }
136
137   int line_starts_with_bar(int line_number)
138   {
139     char *temp = lines[line_number].start;
140
141     while (isspace(*temp))
142       temp--;
143
144     return *temp == '|';
145   }
146
147   void indent_pascal(int line_number, int pascal_indent)
148   {
149     int local_indent;
150     for (local_indent = 0; local_indent < pascal_indent; local_indent++)
151       printf(" ");
152
153     if (!line_starts_with_bar(line_number))
154       printf("  ");
155   }
156
157   void write_to_end_of_line(char *start)
158   {
159     while (*start != '\n')
160       printf("%c", *start++);
161   }
```

```
162
163 int write_pattern(int is_nonterminal)
164 {
165   int printed = 0;
166
167   if (use_ebnf && scan_end - scan_start == 1 &&
168       ( *scan_start == '[' ||
169         *scan_start == ']' ||
170         *scan_start == '{' ||
171         *scan_start == '}' ||
172         *scan_start == '(' ||
173         *scan_start == ')' ||
174         *scan_start == '|'
175       ))
176     printed += printf("%c ", *scan_start);
177   else if (use_dots && *scan_start == '.' && *(scan_start+1) == '.' && *(scan_start+2) == '.')
178     printed += printf("... ");
179   else
180   {
181     char *c;
182     int drop = has_opt_suffix();
183
184     if (drop)
185       scan_end -= 3;
186
187     if (!is_nonterminal)
188       printed += printf("\"");
189
190     for (c = scan_start; c < scan_end; c++)
191       if (is_nonterminal && *c == '-')
192         printed += printf("_");
193       else if (isprint(*c)&& *c != '\"' && *c != '\'' && *c != '\\')
194         printed += printf("%c", *c);
195       else
196       {
197         printed += printf("\\");
198         switch (*c)
199         {
200           case'\a': printed += printf("a"); break;
201           case'\b': printed += printf("b"); break;
202           case'\f': printed += printf("f"); break;
203           case'\n': printed += printf("n"); break;
204           case'\r': printed += printf("r"); break;
205           case'\t': printed += printf("t"); break;
206           case'\v': printed += printf("v"); break;
207           case'\\': printed += printf("\\"); break;
208           case'\'': printed += printf("\'"); break;
209           case'\"': printed += printf("\""); break;
210           default: printed += printf("X%.2X", *c); break;
211         }
212       }
213
214     if (!is_nonterminal)
215       printed += printf("\"");
216
217     if (drop)
218       scan_end += 3;
```

```
219     }
220
221     return printed;
222   }
223
224   int pattern_compare(int index)
225   {
226     char *left_start = scan_start;
227     char *left_end = scan_end;
228     char *right_start = lines[index].lhs_start;
229     char *right_end = lines[index].lhs_end;
230
231     for (;
232          left_start < left_end && right_start < right_end;
233          left_start++, right_start++)
234     {
235       if (*left_start != *right_start)
236         return 0;
237     }
238
239     if (left_start == left_end && right_start == right_end)
240       return 1;
241     else
242       return 0;
243   }
244
245   void write_production(char *start, int in_one_of_rule)
246   {
247     int first = 1;
248
249     printf("  ");
250     scan_end = start;
251     scan();
252     while (*scan_start != '\n')
253     {
254       int is_nonterminal = 0;
255       int drop = has_opt_suffix();
256       int temp;
257
258       if (!first && in_one_of_rule)
259         printf("| ");
260
261       first = 0;
262
263       if (drop)
264         scan_end -=3;
265
266       /* Now look to see if this appears as a LHS nonterminal anywhere */
267       for (temp = 0; temp < linecount; temp++)
268       {
269         if (lines[temp].kind == K_START_RULE ||
270             lines[temp].kind == K_START_RULE_ONE_OF)
271           is_nonterminal |= pattern_compare(temp);
272       }
273
274       if (drop)
275         scan_end +=3;
```

```
276
277       write_pattern(is_nonterminal);
278
279       if (drop)
280         printf("?");
281
282       printf(" ");
283       scan();
284     }
285   }
286
287   int white_prefix_length(int index)
288   {
289     int prefix_length = 0;
290     char *c = lines[index].start;
291
292     while (isspace(*c++))
293       prefix_length++;
294
295     return prefix_length;
296   }
297
298   void help(void)
299   {
300     printf("gramex V2.1 (c) Adrian Johnstone 2006, 2007\n\n"
301            "Usage: gramex [options] source\n\n"
302            "-c   suppress comment (non-rule) lines\n"
303            "-d   treat ... as meta symbol\n"
304            "-e   treat ( ) [ ] { } | as EBNF meta symbols\n"
305            "-i   treat production lines indented or outdented by two or more"
306            " spaces as continuation lines\n"
307            "-p   process Pascal standard file (implies -e)\n"
308            "-s   suppress section numbers at start of line\n"
309           );
310   }
311
312   void test_and_replace(char *current, char *substring, char*replacement)
313   {
314     if (strlen(substring) != strlen(replacement))
315     {
316       printf("Internal programming error:"
317              " substring '%s' and replacement string '%s' lengths differ\n",
318              substring, replacement);
319       exit(1);
320     }
321
322     if (strncmp(current, substring, strlen(substring)) == 0)
323       memcpy(current, replacement, strlen(replacement));
324   }
325
326   void process_pascal(void)
327   {
328     int line_number;
329     char *a_i_string_10206 = "6.1.9 apostrophe_image = '''' .";
330     char *a_i_string_7185 = "6.1.7 apostrophe_image = '''' .";
331
332     /* Inplace translations to ASCII */
```

```
333
334     for (line_number = 0; line_number < linecount - 1; line_number++)
335     {
336       char *current = lines[line_number].start;
337
338   //    printf("\n*** %i ***\n", line_number);
339
340       while (*current != '\n')
341       {
342   //      printf("%c (%i)", *current, *current);
343
344         if (*current == -83)
345           *current = '_';
346         else if (*current == '`')
347         {
348           *current = '\'';
349
350           if ((strncmp(lines[line_number].start, a_i_string_7185, strlen(a_i_string_7185)) == 0 )||
351               (strncmp(lines[line_number].start, a_i_string_10206, strlen(a_i_string_10206)) == 0) )
352             test_and_replace(current, "''''", "'\"' ");
353           else
354             test_and_replace(current, "''''", "'^' ");
355
356           test_and_replace(current, "'\\Gamma'", "'-'     ");
357           test_and_replace(current, "'!'", "'<'");
358           test_and_replace(current, "'?'", "'>'");
359           test_and_replace(current, "'!='", "'<='");
360           test_and_replace(current, "'?='", "'>='");
361           test_and_replace(current, "'=?'", "'=>'");
362           test_and_replace(current, "'!?'", "'<>'");
363           test_and_replace(current, "'?!'", "'><'");
364           test_and_replace(current-1, " '''", "'\\''");
365           test_and_replace(current, "' '", "'_'");
366           test_and_replace(current, "'and then'", "'and_then'");
367           test_and_replace(current, "'or else'", "'or_else'");
368           test_and_replace(current, "'or else'", "'or_else'");
369         }
370         else if (isspace(*(current - 1)) && isspace(*(current + 1)))
371         {
372           if (*current == 'f')
373             *current = '{';
374           else if (*current == 'g')
375             *current = '}';
376           else if (*current == 'j')
377             *current = '|';
378           else if (*current == '?')
379             *current = '>';
380         }
381
382         current++;
383       }
384     }
385
386     /* Annotate production start lines */
387     for (line_number = 0; line_number < linecount - 1; line_number++)
388     {
389       scan_end = lines[line_number].start;
```

```
390
391       scan();
392
393       if (is_section_number())
394       {
395         scan();
396
397         if (is_valid_pascal_nonterminal())
398         {
399           char *temp_scan_start = scan_start;
400           char *temp_scan_end = scan_end;
401
402           scan();
403
404           if ((*scan_start == '=' || *scan_start == '>') && (*(scan_start+1) == '\'' || isspace(*(scan_star
405           {
406             lines[line_number].kind = K_PASCAL_START_RULE;
407             lines[line_number].lhs_start = temp_scan_start;
408             lines[line_number].lhs_end = temp_scan_end;
409
410             if (line_ends_with_period(line_number))
411               lines[line_number].kind = K_PASCAL_START_END_RULE;
412           }
413         }
414       }
415     }
416
417     /* Annotate continuation lines */
418     for (line_number = 1; line_number < linecount - 1; line_number++)
419     {
420       if (lines[line_number].kind == K_COMMENT) /* presently unlabelled */
421       {
422         if (lines[line_number - 1].kind == K_PASCAL_START_RULE ||
423             lines[line_number - 1].kind == K_PASCAL_CONTINUATION)
424         {
425           if (line_ends_with_period(line_number))
426             lines[line_number].kind = K_PASCAL_END_RULE;
427           else
428             lines[line_number].kind = K_PASCAL_CONTINUATION;
429         }
430       }
431     }
432   }
433
434   void process_c(void)
435   {
436     int in_rule = 0;
437     int temp;
438
439     /* Annotate start lines */
440     for (temp = 0; temp < linecount - 1; temp++)
441     {
442       scan_end = lines[temp].start;
443
444       scan();
445
446       if (initial_alpha_has_colon_suffix())
```

```
447       {
448         lines[temp].kind = K_START_RULE;
449         lines[temp].lhs_start = scan_start;
450         lines[temp].lhs_end = scan_end - 1; /* drop trailing colon */
451
452         scan();
453
454           /* Check for 'one of' */
455           if (((scan_end - scan_start) == 3) && *(scan_start) == 'o' &&
456                                                 *(scan_start+1) == 'n' &&
457                                                 *(scan_start+2) == 'e')
458           {
459             scan();
460             if (((scan_end - scan_start) == 2) && *(scan_start) == 'o' &&
461                                                   *(scan_start+1) == 'f')
462               lines[temp].kind = K_START_RULE_ONE_OF;
463             scan();
464           }
465
466           if (*scan_start != '\n')
467             lines[temp].kind = K_MAYBE_TYPO;
468         }
469         else
470         {
471           scan();
472           if ((scan_end - scan_start == 1) && *scan_start == ':')
473             lines[temp].kind = K_MAYBE_TYPO;
474         }
475     }
476
477     /* Annotate productions */
478     for (temp = 0; temp < linecount - 1; temp++)
479     {
480
481       if (lines[temp].kind == K_START_RULE ||
482           lines[temp].kind == K_START_RULE_ONE_OF)
483         in_rule = 1;
484
485       if (lines[temp].kind == K_BLANK)
486         in_rule = 0;
487
488       if (in_rule && (lines[temp].kind == K_COMMENT ||
489           lines[temp].kind == K_MAYBE_TYPO))
490       {
491         lines[temp].kind = K_PRODUCTION;
492
493         if (lines[temp - 1].kind == K_START_RULE)
494           first_production_indent = white_prefix_length(temp);
495
496         if (use_indentation_as_continuation &&
497             ((white_prefix_length(temp) > (first_production_indent + 1)) |
498              (white_prefix_length(temp) < (first_production_indent - 1))
499             ))
500           lines[temp - 1].kind = K_PRODUCTION_CONTINUED;
501       }
502     }
503
```

```
504     /* Annotate final productions */
505     for (temp = 0; temp < linecount - 1; temp++)
506       if (lines[temp].kind == K_PRODUCTION &&
507           !(lines[temp+1].kind == K_PRODUCTION ||
508             lines[temp+1].kind == K_PRODUCTION_CONTINUED
509            )
510           )
511         lines[temp].kind = K_END_RULE;
512   }
513
514   int main(int argc, char *argv[])
515   {
516     FILE *f;
517     char *filename;
518     unsigned temp;
519     char *tempstring;
520     int in_one_of_rule = 0;
521     unsigned charcount;
522     int pascal_indent;
523     int nonterminal_length;
524
525     if (argc < 2)
526     {
527       help();
528       return 1;
529     }
530
531     for (temp = 1; temp < argc; temp++)
532     {
533       if (*argv[temp] == '-')  /* option */
534         switch (*(argv[temp] + 1))
535         {
536           case 'c': suppress_comments = 1; break;
537           case 'd': use_dots = 1; break;
538           case 'e': use_ebnf = 1; break;
539           case 'i': use_indentation_as_continuation = 1; break;
540           case 'p': pascal_mode = 1; use_ebnf = 1; break;
541           case 's': suppress_section_numbers = 1; break;
542           default:
543             printf("Unknown option -%c\n\n", *(argv[temp] + 1));
544             help();
545             return 1;
546         }
547       else
548         filename = argv[temp];
549     }
550
551     if ((f = fopen(filename, "r")) == NULL)
552     {
553       printf("Unable to open input file '%s' for read\n\n", filename);
554       help();
555       return 0;
556     }
557
558     /* Size the file and allocate the buffer */
559     charcount = 0;
560     while (charcount++, getc(f) != EOF)
```

```
561       ;
562     rewind(f);
563
564     if ((buffer = (char*) malloc(charcount+2)) == NULL)
565     {
566       printf("Unable to allocate buffer\n");
567       return 0;
568     }
569
570     /* Load input buffer */
571     charcount = 0;
572     while (1)
573     {
574       int ch = getc(f);
575
576       if (ch == EOF)
577       {
578         if (buffer[charcount - 1] != '\n')
579           buffer[charcount++] = '\n';  /* tack on a trailing \n if missing */
580         break;
581       }
582       else
583         buffer[charcount++] = (char) ch;
584     }
585
586     /* Count lines */
587     for (temp = 0; temp < charcount; temp++)
588       if (buffer[temp] == '\n')
589         linecount++;
590
591     /* Allocate lines buffer */
592     lines = (struct line_struct*)
593             calloc(linecount + 2, sizeof(struct line_struct));
594
595     if (lines == NULL)
596     {
597       printf("Unable to allocate lines buffer\n");
598       return 1;
599     }
600
601     /* Load lines start entries */
602     linecount = 0;
603     lines[linecount++].start = &buffer[0];
604
605     for (temp = 0; temp < charcount; temp++)
606       if (buffer[temp] == '\n')
607         lines[linecount++].start = &buffer[temp+1];
608
609     /* Annotate blank lines */
610     for (temp = 0; temp < linecount - 1; temp++)
611     {
612       int is_blank = 1;
613
614       for (tempstring = lines[temp].start; *tempstring != '\n'; tempstring++)
615         if (!isspace(*tempstring))
616           is_blank = 0;
617
```

```
618       if (is_blank)
619         lines[temp].kind = K_BLANK;
620     }
621
622     if (suppress_section_numbers)
623     for (temp = 0; temp < linecount - 1; temp++)
624     {
625       scan_end = lines[temp].start;
626
627       scan();
628
629       if (is_section_number())
630         lines[temp].kind = K_SECTION_HEADING;
631     }
632
633
634     if (pascal_mode)
635       process_pascal();
636      else
637        process_c();
638
639     /* Output lines */
640     printf("// Generated by gramex V2.1 from '%s' on "
641            __DATE__ " at " __TIME__ "\n",
642            filename);
643     printf("// Command line:");
644     for (temp = 0; temp < argc; temp++)
645       printf(" %s", argv[temp]);
646
647     printf("\n\n");
648
649     for (temp = 0; temp < linecount - 1; temp++)
650     {
651       char *c;
652
653       scan_end = lines[temp].start;
654
655       switch (lines[temp].kind)
656       {
657         case K_COMMENT:
658           if (!suppress_comments)
659           {
660             printf("// ");
661             for (c = lines[temp].start; *c != '\n'; c++)
662               printf("%c", *c);
663             printf("\n");
664           }
665           break;
666
667         case K_MAYBE_TYPO:
668           if (!suppress_comments)
669           {
670             printf("//?? ");
671             for (c = lines[temp].start; *c != '\n'; c++)
672               printf("%c", *c);
673             printf("\n");
674           }
```

```
675           break;
676
677         case K_START_RULE:
678           scan();
679           write_pattern(1);
680           printf("\n");
681           in_one_of_rule = 0;
682           break;
683
684         case K_PASCAL_START_END_RULE:
685         case K_PASCAL_START_RULE:
686           pascal_indent = 0;
687
688           printf("\n");
689
690           scan();
691
692           if (!suppress_section_numbers)
693           {
694             pascal_indent += printf("(*");
695             pascal_indent += write_pattern(1);
696             pascal_indent += printf("*) ");
697
698             while (pascal_indent < 14)
699               pascal_indent += printf(" ");
700           }
701
702           scan();
703           nonterminal_length = write_pattern(1);
704           pascal_indent += nonterminal_length;
705
706           pascal_indent += 1; /* allow for position of = in rule */
707
708           in_one_of_rule = 0;
709
710           if (line_ends_with_equals(temp))
711             pascal_indent = pascal_indent - nonterminal_length + 2;
712
713           write_to_end_of_line(scan_end);
714           printf("\n");
715
716           break;
717
718         case K_PASCAL_CONTINUATION:
719
720           indent_pascal(temp, pascal_indent);
721           write_to_end_of_line(scan_end);
722           printf("\n");
723           break;
724
725         case K_PASCAL_END_RULE:
726           indent_pascal(temp, pascal_indent);
727           write_to_end_of_line(scan_end);
728           printf("\n");
729           break;
730
731         case K_START_RULE_ONE_OF:
```

```
732           scan();
733           write_pattern(1);
734           printf("\n");
735           in_one_of_rule = 1;
736           break;
737
738         case K_END_RULE:
739           write_production(lines[temp].start, in_one_of_rule);
740           printf(";\n");
741           if (suppress_comments)
742             printf("\n");
743           break;
744
745         case K_PRODUCTION:
746           write_production(lines[temp].start, in_one_of_rule);
747           printf("|\n");
748           break;
749
750         case K_PRODUCTION_CONTINUED:
751           write_production(lines[temp].start, in_one_of_rule);
752           printf("\n");
753           break;
754
755         case K_BLANK:
756           if (!suppress_comments)
757             printf("\n");
758           break;
759       }
760     }
761
762     return 0;
763   }
764
```