APPENDIX C

LLGen User Manual

LLGen accepts a context-free grammar specification and produces tables for parsing the language so specified. It will produce tables for any LL(1) grammar and provides a simple conflict resolution mechanism for grammars that are not LL(1). LLGen is a subset of FMQ, an LL(1) parser/error-corrector generator. FMQ was written by Jon Mauney.

This report describes a tool for language translation—specifically, for parsing context-free languages. LLGen is a table generator. It accepts an LL(1) grammar specified in the format described below and produces tables that can be used for parsing. LLGen also provides an interface to user-supplied semantic actions through semantic action numbers. These numbers are specified in the input to LLGen and appear in the resulting tables.

A typical session with LLGen might go as follows:

(1) Create a file specifying the desired grammar.

(2) Run LLGen, directing the grammar file to standard input. LLGen will send optional output and error messages, if any, to the terminal (or whatever the standard output is) and create a file, ptableout or ptablebin (see the following section on output).

(3) If the grammar is not acceptable to LLGen, repeat 1 and 2.

(4) Use ptableout or ptablebin in an LL(1) parser driver.

C.1 Input to LLGen

The input to LLGen has three main sections: options desired for the run, terminal symbols of the grammar, and production rules of the grammar. The general form of the input is

```
    <comments>
*fmq
    <options>
*define
    <constant definitions>
*terminals
    <terminal specifications>
*productions
    <production specifications>
*end
    <comments>
```

An example appears in Figure C.2.

In the following, *symbol* will refer to a symbol in the grammar for which tables are to be generated, and *token* will refer to an entity in the input to LLGen.

The input to LLGen is divided into tokens by three simple rules:

- All tokens must be separated by one or more blanks, tabs, or ends of line.
- Tokens may not contain blanks or tabs, unless the token is surrounded by angle brackets, **<** and **>.** Tokens may not run across line boundaries.
- If a token begins with **<,** then it must end with **>**.

That is, everything in the input—option names, reserved words, grammar symbols—must be surrounded by white space. Angle brackets may be used when a symbol contains white space, but they are special if and only if the first character is **<.** Angle brackets appearing in any other circumstances are legal but not special (a warning will be issued in such cases). The only other character endowed with special properties is **#**, which begins all *action symbols.* Action symbols consist of **#** followed immediately by an unsigned integer or a defined constant (as described below).

Upper- and lower-case letters are considered distinct; however, the reserved tokens and the options are recognized in either case or a mixture. The examples shown in Figure C.1 illustrate the above rules.

The following tokens are reserved:

```
*fmq   *define   *terminals   *productions   *end
::=    ...       <Goal>       $$$
```

End of line is required as a terminator for specifications of terminals, productions, and constant definitions as described below. The input to LLGen is otherwise free-format.

Anything before ***fmq** or after ***end** will be considered a comment and will be ignored. However, comments must not contain any of the above reserved tokens. Following ***fmq** is a list of zero or more options, separated, as always, by blanks, tabs, or ends of line. All options have the form of switches and are enabled by including the name in the option list. An enabled

token	**comments**
ABC	OK
abc	OK, different from **ABC**
123	OK
< Expr >	OK
<id list>	OK
<>	OK
&:=	OK
"<"	Legal, gets a warning
—>	Legal, gets a warning
<not<>equal>	Legal, gets a warning
much<<lessthan	Legal, gets a warning
<LHS>::=<RHS>	Legal, gets a warning. This is one token, not three.
2<two tokens>	Legal, two tokens, two warnings
	(**<** is special only if it appears first)
∗**fmq**	Reserved
∗**FMQ**	Reserved, same as ∗**fmq**
∗**Fmq**	Reserved, same as ∗**fmq**
#13	Legal, action symbol for routine 13
#add	Legal, add should have been defined
	in the ∗**define** section
∗**fmq**∗**terminals**	legal, one token, no warning
<=	Illegal, no closing bracket
<	Illegal, no closing bracket
<bad>token	Ditto, (**>** must be followed by space)

Figure C.1 Examples of LLgen Tokens

option may be disabled by placing **no** before the name, without a space; thus, to prevent construction of parse tables, use **noparsetable**. All options are initially disabled, except for **checkreduce** and **text**. Options are recognized in upper or lower case. The available options are

(1) **bnf**
Print the grammar rules.

(2) **first**
Print the **First** sets for all nonterminals.

(3) **follow**
Print the **Follow** sets of all nonterminals.

(4) **parsetable**
Print the parse action table in tabular form. A number in the table indicates prediction of that production; blank entries indicate error. This table can be quite large.

(5) **checkreduce**
Check whether the grammar is reduced; report all symbols that cannot

produce a terminal string and those that are not reachable from the start symbol. If the grammar is not reduced, and checkreduce is enabled, tables will not be produced. **`Checkreduce`** is normally enabled.

(6) **`resolve`**
If the grammar given is not LL(1), generate the parse tables anyway, and resolve parse conflicts pairwise in favor of the production that appeared earlier in the input. This option should be used with caution; see discussion under "Error Handling within LLGen." If **`resolve`** is not enabled, computation of parse tables will be suppressed in the presence of parse conflicts.

(7) **`shortline`**, **`longline`**
Control the length of printed lines in human-oriented output (**`vocab`**, **`parsetable`**, etc.) **`Shortline`** causes lines to be less than 80 characters (suitable for a screen), and **`longline`** is 132 (for a printer). **`Shortline`** is synonymous with **`nolongline`**, and vice versa. The default is **`shortline`**.

(8) **`statistics`**
Print assorted statistics on the grammar.

(9) **`vocab`**
Print the symbols of the language.

(10) **`text`**, **`binary`**
The tables created by LLGen can be written as text (file of characters), as binary (file of integers), or both. Text output is written on the file **`ptableout`**; binary is written on **`ptablebin`**. Binary files tend to be larger, at least on a 32-bit machine (about 30% larger on the VAX under UNIX^{tm}), but are usually faster to read. The default is **`text`** and **`nobinary`**.

The constant definition section is optional. If present, it begins with the reserved token ∗**`define`** and consists of a list of definitions, each on a separate line. Each definition has the form

<const name> <integer value>

in which <const name> is a token as described above, and <integer value> is an unsigned integer (i.e., a token containing only digits). This constant can then be used whenever an integer constant is called for: in subsequent constant definitions and for semantic routine numbers. Note that this feature is not as nice as it seems at first, because the output listing of LLGen will use the numeric value, not the constant name.

The reserved token ∗**`terminals`** begins the list of terminal symbols. The terminal specification section consists of a list of such specifications, each on a separate line. All terminals *must* appear in this list.

The token ∗**`productions`** separates the terminals from the productions. The productions are specified by a list of rules, each on a separate line.

Specification of one production has the form

<lhs> **::=** <rhs>

Either <lhs> or <rhs> may be absent. <lhs> is one token representing a nonterminal symbol. If it is absent, the <lhs> of the preceding production is used. <rhs> is a string of tokens, containing the grammar symbols of the production and action symbols indicating semantic routines to be called when the appropriate point in the production is reached. An action symbol consists of a # followed by an unsigned integer or defined constant, without intervening blanks. If <rhs> is absent or contains only action symbols, then <lhs> derives the null string. <rhs> may be continued on subsequent lines by beginning those lines with the reserved token “**"..."**” (only productions may be so continued).

The productions are terminated by *__end__. After all of the productions have been processed, the augmenting production is added. Two symbols, **<Goal>** and **$$$**, and one production

<Goal> ::= <S> **$$$**

are added to the grammar, where <S> is the left-hand side of the first production specified, **<Goal>** is the start symbol, and **$$$** the end-marker.

C.2 Output from LLGen

The output controlled by the above options is written to the standard Pascal file **output**. In addition, files containing tables are created. The tables for parsing are written to **ptableout** (text) or **ptablebin** (binary). These files may be assigned or redirected, depending on the operating system.

The formats of the output files are the same, whether they are created as **text** or **binary** files. In **binary** files, character values are written as ord of the character.

The files **ptableout** and **ptablebin** contain the following tables: the right-hand sides of all productions, the parser action table, a list of symbols that can derive the null string, and symbolic representations of all symbols in the grammar.

The format of the file is

header line:
The first line gives the sizes of the various tables. It contains the number of terminals in the language (numterms), the number of symbols in the grammar (numsymbols), the number of productions in the grammar (numprods), the size of the character string for the symbol images (stringsize), and a flag indicating whether error-repair tables were produced. The value of the flag is a character **T** if the repair tables were produced, and **F** if not.

productions:
The right-hand sides of all the productions are given; all right-hand sides are *reversed* and can be pushed onto the parse stack in the order given. Each production consists of a length, followed by the corresponding number of symbols. Action symbols are included in the right-hand sides, encoded as the negative of the number given in the grammar.

parse table:
The LL(1) predictions for each nonterminal symbol are given by the parse table. The predictions for each nonterminal are stored in a list of pairs of integers. The first pair in a list is a zero followed by the number of the nonterminal symbol. The subsequent pairs have the form

```
terminal prediction
```

in which `prediction` is the production predicted when `terminal` appears in the lookahead. The list is terminated by the start of the next list. The table is terminated by 0 0.

lambda productions:
Lambda productions are those that have no symbols (except action symbols) on the right-hand side. The list consists of a length, `n`, followed by `n` integers representing the production numbers. (This table isn't needed for ordinary LL(1) parsing, but it is useful in avoiding unnecessary parser moves when error repair is done.)

string table:
The symbol table information is in two parts. First is an index, consisting of `numsymbols` pairs of integers. The first integer of each pair is the starting point of the symbol in the character string, and the second is the length of the symbol. Following the index are `stringsize` characters, 132 per line.

C.3 Error Handling within LLGen

Syntax errors in the input to LLGen will be handled by an error recovery routine. Table generation will be suppressed if errors are present in the input.

An attempt to use the symbols **`<Goal>`** and **`$$$`** (the start symbol and the end-marker) will be treated as a syntax error.

All terminals must be listed in the **`*terminals`** section. If any terminal is not listed or if a nonterminal does not appear on the left of any production, the symbol will be flagged, and no tables will be generated. Similarly, a symbol declared as a terminal may not appear on the left of a production.

If the grammar specified is not LL(1), all conflicts will be reported. If the option **`resolve`** is enabled, productions will be given precedence in the order of appearance (the first production specified has highest precedence). Thus the dangling **else** of Pascal and other languages can be parsed by

```
<if stmt>     ::= if <expr> then <stmt> <else part>
<else part>   ::= else <stmt>
              ::=
```

The conflict will be resolved in favor of the first form of the statement, matching the **else** with the most recent **if**.

This resolution mechanism should be used with caution. Conflicts must be carefully examined to ensure that the parse action taken is the action desired. For example, reversing the order of the above **<else part>** productions would be perfectly acceptable to LLGen but would have a disastrous effect. When **else** appears in the lookahead, the parse action taken would be always to predict **<else part>** deriving the null string; the **else** would never be accepted.

If the grammar specified proves too large for the limits of LLGen, the program will print a message describing the limit that was exceeded and will terminate. LLGen must then be recompiled with increased limits. Normally, exceeding one limit suggests that others will also be exceeded, and increasing them all at once will save on recompilations. Notice though that LLGen processes in order terminals, productions, and parse table. Therefore, if the number of productions in the grammar is exceeded, the number of terminals must be within limits, as they have been completely processed already. Some of the dimensions of a particular grammar are easy to discover; others must be tackled by rule of thumb and trial. Easily determined are the number of terminals, number of symbols (terminals+nonterminals) and number of productions. Less simple, but still easy to estimate, are the total number of symbols in the productions and the total number of characters in all the distinct symbols.

C.4 Using LLGen

The grammar specification is read from the standard input file, and human-oriented output is written to standard output. Parse tables produced are written to **ptableout** or **ptablebin**. The above names are the internal names as declared in the program header and may be modified by the system environment in which the program is run.

Figure C.2 illustrates a sample LLGen input specification; Figure C.3 illustrates the ptableout file that would be produced.

```
Grammar for DCL, a desk calculator language

(the action symbols here don't mean anything,
 they just illustrate how they might be used)

*fmq
statistics
checkreduce
```

```
vocab bnf
text  binary
*terminals
    id
    constant
    end
    ;
    :=
    (
    )
    +
    -
    *
    /
    write
    read
    ,

*productions
<prog>            ::= <st list> end
<st list>         ::= <st> #1 <st list tail>
<st list tail>    ::= ; <st list>
                  ::=
<st>              ::= id #2 := <expr> #5
                  ::= read  #3 ( <id list> #6 #14 )
                  ::= write  #4 ( <expr list> #6 )
<expr>            ::= <term> <e tail>
<e tail>          ::= + <term> #20 <e tail>
<e tail>          ::= - <term> #20 <e tail>
                  ::=
<term>            ::= <primary> <t tail>
<t tail>          ::= * <primary> #21 <t tail>
<t tail>          ::= / <primary> #21 <t tail>
                  ::=
<primary>         ::= - <primary> #7
<primary>         ::= ( <expr> )
                  ::= id #8
                  ::= constant #9
<expr list>       ::= <expr> #30 <e list tail>
<e list tail>     ::= , <expr list>
                  ::=
<id list>         ::= id #15 <id list tail>
<id list tail>    ::= , <id list>
                  ::=
*end
```

Figure C.2 Sample LLGen Input

```
 15 29 26 243 F
2 3 17 3 19 −1 18 2 17 4 0 5 −5 20 5 −2 1 7 7
−14 −6 21 6 −3 13 6 7 −6 22 6 −4 12 2 24
23 4 24 −20 23 8 4 24 −20 23 9 0 2 26
25 4 26 −21 25 10 4 26 −21 25 11 0 3 −7 25 9 3
7 20 6 2 −8 1 2 −9 2
3 27 −30 20 2 22 14 0 3 28 −15 1 2 21 14 0 2 15 16
0 16 13 1 12 1 1 1 0 17 13 2 12 2 1 2 0 18 12 7 13 6 1 5
0 19 3 4 4 3 0 20 9
8 6 8 2 8 1 8 0 21 1 23 0 22 9 20 6 20 2 20 1 20 0 23 9
12 6 12 2 12 1 12 0 24 14 11 7 11 4 11 3 11 9 10 8 9 0 25
2 19 1 18 6 17 9 16 0 26 14 15 9 15 8 15 7 15 4 15 3 15 11
14 10 13 0 27 7 22 14 21 0 28 7 25 14 24 0 29 13 26 12 26
1 26 0 0 5 4 11 15 22 25 95 2 97 8 105 3 108 1 109 2 111 1
112 1 113 1 114 1 115 1 116 1 117 5 122 4 126
1 7 3 127 6 133 9 142 4 146 14 160 6 166 9 175 11 186 6 192
8 200 9 209 8 217 13 230 14 1 6
<Goal>$$$ grammarforDCL(theactionsymbolsheredon'tmeananything,
theyjustillustratehowtheymightbeidconstantend;:=()+−*/wr
iteread,<prog><st list><st ><st list tail>< expr><id list><exp
r list><term><e tail><primary><t tail><e list tail><id list tail>
```

Figure C.3 **`ptableout`** Produced by LLGen Given the Input File of Figure C.2