

NAME

LLnextgen – an Extended-LL(1) parser generator

SYNOPSIS

LLnextgen [*OPTIONS*] [*FILES*]

DESCRIPTION

LLnextgen is a (partial) reimplementaion of the **LLgen** ELL(1) parser generator created by D. Grune and C.J.H. Jacobs (note: this is not the same as the **LLgen** parser generator by Fischer and LeBlanc). It takes an EBNF-like description of the grammar as input(s), and produces a parser in C.

Input files are expected to end in .g. The output files will have .g removed and .c and .h added. If the input file does not end in .g, the extensions .c and .h will simply be added to the name of the input file. Output files can also be given a different base name using the option `--base-name` (see below).

OPTIONS

LLnextgen accepts the following options:

-c, --max-compatibility

Set options required for maximum source-level compatibility. This is different from running as **LLgen**, as all extensions are still allowed. LLreissue and the prototypes in the header file are still generated. This option turns on the `--llgen-arg-style`, `--llgen-escapes-only` and `--llgen-output-style` options.

-e, --warnings-as-errors

Treat warnings as errors.

-Enum, --error-limit=num

Set the maximum number of errors, before **LLnextgen** aborts. If *num* is set 0, the error limit is set to infinity. This is to override the error limit option specified in the grammar file.

-h[which], --help[=which]

Print out a help message, describing the options. The optional *which* argument allows selection of which options to print. *which* can be set to all, depend, error, and extra.

-V, --version

Print the program version and copyright information, and exit.

-v[level], --verbose[=level]

Increase (without explicit level) or set (with explicit level) the verbosity level. **LLnextgen** uses this option differently than **LLgen**. At level 1, **LLnextgen** will output traces of the conflicts to standard error. At level 2, **LLnextgen** will also write a file named LL.output with the rules containing conflicts. At level 3, **LLnextgen** will include the entire grammar in LL.output.

LLgen will write the LL.output file from level 1, but cannot generate conflict traces. It also has an intermediate setting between **LLnextgen** levels 2 and 3.

-w[warnings], --suppress-warnings[=warnings]

Suppress all or selected warnings. Available warnings are: arg-separator, option-override, unbalanced-c, multiple-parser, eofile, unused[:<identifier>], datatype and unused-retval. The unused warning can suppress all warnings about unused tokens and non-terminals, or can be used to suppress warnings about specific tokens or non-terminals by adding a colon and a name. For example, to suppress warning messages about FOO not being used, use `-wunused:FOO`. Several comma separated warnings can be specified with one option on the command line.

--abort

Generate the LLabort function.

--base-name=name

Set the base name for the output files. Normally **LLnextgen** uses the name of the first input file without any trailing .g as the base name. This option can be used to override the default. The files created will be *name.c* and *name.h*. This option cannot be used in combination with `--llgen-output-style`.

--depend[=*modifiers*]

Generate dependency information to be used by the **make**(1) program. The modifiers can be used to change the make targets (targets:<targets>, and extra-targets:<targets>) and the output (file:<file>). The default are to use the output names as they would be created by running with the same arguments as targets, and to output to standard output. Using the targets modifier, the list of targets can be specified manually. The extra-targets modifier allows targets to be added to the default list of targets. Finally, the phony modifier will add phony targets for all dependencies to avoid **make**(1) problems when removing or renaming dependencies. This is like the **gcc**(1) -MP option.

--depend-cpp

Dump all top-level C-code to standard out. This can be used to generate dependency information for the generated files by piping the output from **LLnextgen** through the C preprocessor with the appropriate options.

--dump-lexer-wrapper

Write the lexer wrapper function to standard output, and exit.

--dump-llmessage

Write the default LLmessage function to standard output, and exit.

--dump-tokens[=*modifier*]

Dump %token directives for unknown identifiers that match the **--token-pattern** pattern. The default is to generate a single %token directive with all the unknown identifiers separated by comma's. This default can be overridden by *modifier*. The modifier *separate* produces a separate %token directive for each identifier, while *label* produces a %label directive. The text of the label will be the name of the identifier. If the *label* modifier and the **--lowercase-symbols** option are both specified the label will contain only lowercase characters.

Note: this option is not always available. It requires the POSIX regex API. If the POSIX regex API is not available on your platform, or the **LLnextgen** binary was compiled without support for the API, you will not be able to use this option.

--extensions=*list*

Specify the extensions to be used for the generated files. The list must be comma separated, and should not contain the . before the extension. The first item in the list is the C source file and the second item is the header file. You can omit the extension for the C source file and only specify the extension for the header file.

--generate-lexer-wrapper[=*yes/no*]

Indicate whether to generate a wrapper for the lexical analyser. As **LLnextgen** requires a lexical analyser to return the last token returned after detecting an error which requires inserting a token to repair, most lexical analysers require a wrapper to accommodate **LLnextgen**. As it is identical for almost each grammar, **LLnextgen** can provide one. Use **--dump-lexer-wrapper** to see the code. If you do specify this option **LLnextgen** will generate a warning, to help remind you that a wrapper is required.

If you do not want the automatically generate wrapper you should specify this option followed by **=no**.

--generate-llmessage

Generate an *LLmessage* function. **LLnextgen** requires programs to provide a function for informing the user about errors in the input. When developing a parser, it is often desirable to have a default *LLmessage*. The provided *LLmessage* is very simple and should be replaced by a more elaborate one, once the parser is beyond the first testing phase. Use **--dump-llmessage** to see the code. This option automatically turns on **--generate-symbol-table**.

--generate-symbol-table

Generate a symbol table. The symbol table will contain strings for all tokens and character literals. By default, the symbol table contains the token name as specified in the grammar. To change the string, for both tokens and character literals, use the %label directive.

--gettext[=*macro,guard*]

Add gettext support. A macro call is added around symbol table entries generated from %label directives. The macro will expand to the string itself. This is meant to allow **xgettext(1)** to extract the strings. The default is `N_`, because that is what most people use. A guard will be included such that compilation without gettext is possible by not defining the guard. The guard is set to `USE-NLS` by default. Translations will be done automatically in `LLgetSymbol` in the generated parser through a call to `gettext`.

--keep-dir

Do not remove directory component of the input file-name when creating the output file-name. By default, outputs are created in the current directory. This option will generate the output in the directory of the input.

--llgen-arg-style

Use semicolons as argument separators in rule headers. **LLnextgen** uses comma's by default, as this is what ANSI C does.

--llgen-escapes-only

Only allow the escape sequences defined by **LLgen** in character literals. By default **LLnextgen** also allows `\a`, `\v`, `\?`, `\"`, and hexadecimal constants with `\x`.

--llgen-output-style

Generate one `.c` output per input, and the files `Lpars.c` and `Lpars.h`, instead of one `.c` and one `.h` file based on the name of the first input.

--lowercase-symbols

Convert the token names used for generating the symbol table to lower case. This only applies to tokens for which no %label directive has been specified.

--no-allow-label-create

Do not allow the %label directive to create new tokens. Note that this requires that the token being labelled is either a character literal or a %token directive creating the named token has preceded the %label directive.

--no-arg-count

Do not check argument counts for rules. **LLnextgen** checks whether a rule is used with the same number of arguments as it is defined. **LLnextgen** also checks that any rules for which a %start directive is specified, the number of arguments is 0.

--no-cof-zero

Do not use 0 as end-of-file token. **(f)lex(1)** uses 0 as the end-of-file token. Other lexical-analyser generators may use `-1`, and may use 0 for something else (e.g. the nul character).

--no-init-llretval

Do not initialise **LLretval** with 0 bytes. Note that you have to take care of initialisation of **LLretval** yourself when using this option.

--no-line-directives

Do not generate `#line` directives in the output. This means all errors will be reported relative to the output file. By default **LLnextgen** generates `#line` directives to make the C compiler generate errors relative to the **LLnextgen** input file.

--no-llreissue

Do not generate the `LLreissue` variable, which is used to indicate when a token should be reissued by the lexical analyser.

--no-prototypes-header

Do not generate prototypes for the parser and other functions in the header file.

--not-only-reachable

Do not only analyse reachable rules. **LLnextgen** by default does not take unreachable rules into account when doing conflict analysis, as these can cause spurious conflicts. However, if the

unreachable rules will be used in the future, one might already want to be notified of problems with these rules. **LLgen** by default does analyse unreachable rules.

Note: in the case where a rule is unreachable because the only alternative of another reachable rule that mentions it is never chosen (because of a `%avoid` directive), the rule is still deemed reachable for the analysis. The only way to avoid this behaviour is by doing the complete analysis twice, which is an excessive amount of work to do for a very rare case.

--reentrant

Generate a reentrant parser. By default, **LLnextgen** generates non-reentrant parsers. A reentrant parser can be called from itself, but not from another thread. Use `--thread-safe` to generate a thread-safe parser.

Note that when multiple parsers are specified in one grammar (using multiple `%start` directives), and one of these parsers calls another, either the `--reentrant` option or the `--thread-safe` option is also required. If these parsers are only called when none of the others is running, the option is not necessary.

Use only in combination with a reentrant lexical analyser.

--show-dir

Show directory names of source files in error and warning messages. These are usually omitted for readability, but may sometimes be necessary for tracing errors.

--thread-safe

Generate a thread-safe parser. Thread-safe parsers can be run in parallel in different threads of the same program. The interface of a thread-safe parser is different from the regular (and then reentrant) version. See the detailed manual for more details.

--token-pattern=*pattern*

Specify a regular expression to match with unknown identifiers used in the grammar. If an unknown identifier matches, **LLnextgen** will generate a token declaration for the identifier. This option is primarily implemented to aid in the first stages of development, to allow for quick testing for conflicts without having to specify all the tokens yet. A list of tokens can be generated with the `--dump-tokens` option.

Note: this option is not always available. It requires the POSIX regex API. If the POSIX regex API is not available on your platform, or the **LLnextgen** binary was compiled without support for the API, you will not be able to use this option.

By running **LLnextgen** using the name **LLgen**, **LLnextgen** goes into **LLgen**-mode. This is implemented by turning off all default extra functionality like *LLreissue*, and disallowing all extensions to the **LLgen** language. When running as **LLgen**, **LLnextgen** accepts the following options from **LLgen**:

- a** Ignored. **LLnextgen** only generates ANSI C.
- hnum** Ignored. **LLnextgen** leaves optimisation of jump tables entirely up to the C-compiler.
- j[num]** Ignored. **LLnextgen** leaves optimisation of jump tables entirely up to the C-compiler.
- l[num]** Ignored. **LLnextgen** leaves optimisation of jump tables entirely up to the C-compiler.
- v** Increase the verbosity level. See the description of the `-v` option above for details.
- w** Suppress all warnings.
- x** Ignored. **LLnextgen** will only generate token sets in LL.output. The extensive error-reporting mechanisms in **LLnextgen** make this feature obsolete.

LLnextgen cannot create parsers with non-correcting error-recovery. Therefore, using the `-n` or `-s` options will cause **LLnextgen** to print an error message and exit.

COMPATIBILITY WITH LLGEN

At this time the basic **LLgen** functionality is implemented. This includes everything apart from the extended user error-handling with the `%onerror` directive and the non-correcting error-recovery.

Although I've tried to copy the behaviour of **LLgen** accurately, I have implemented some aspects slightly differently. The following is a list of the differences in behaviour between **LLgen** and **LLnextgen**:

- **LLgen** generated both K&R style C code and ANSI C code. **LLnextgen** only supports generation of ANSI C code.
- There is a minor difference in the determination of the default choices. **LLnextgen** simply chooses the first production with the shortest possible terminal production, while **LLgen** also takes the complexity in terms of non-terminals and terms into account. There is also a minor difference when there is more than one shortest alternative and some of them are marked with %avoid. Both differences are not very important as the user can specify which alternative should be the default, thereby circumventing the differences in the algorithms.
- The default behaviour of generating one output C file per input and Lpars.c and Lpars.h has been changed in favour of generating one .c file and one .h file. The rationale given for creating multiple output files in the first place was that it would reduce the compilation time for the generated parser. As computation power has become much more abundant this feature is no longer necessary, and the difficult interaction with the make program makes it undesirable. The **LLgen** behaviour is still supported through a command-line switch.
- in **LLgen** one could have a parser and a %first macro with the same name. **LLnextgen** forbids this, as it leads to name collisions in the new file naming scheme. For the old **LLgen** file naming scheme it could also easily lead to name collisions, although they could be circumvented by not mentioning the parser in any of the C code in the .g files.
- **LLgen** names the labels it generates L_X, where X is a number. **LLnextgen** names these LL_X.
- **LLgen** parsers are always reentrant. As this feature is not used very often, **LLnextgen** parsers are non-reentrant unless the option **--reentrant** is used.

Furthermore, **LLnextgen** has many extended features, for easier development.

BUGS

If you think you have found a bug, please check that you are using the latest version of **LLnextgen** [<http://os.ghalkes.nl/LLnextgen>]. When reporting bugs, please include a minimal grammar that demonstrates the problem.

AUTHOR

G.P. Halkes <llnextgen@ghalkes.nl>

COPYRIGHT

Copyright © 2005-2008 G.P. Halkes

LLnextgen is licensed under the GNU General Public License version 3.

For more details on the license, see the file COPYING in the documentation directory. On Un*x systems this is usually /usr/share/doc/LLnextgen-0.5.5.

SEE ALSO

LLgen(1), **bison(1)**, **yacc(1)**, **lex(1)**, **flex(1)**.

A detailed manual for **LLnextgen** is available as part of the distribution. It includes the syntax for the grammar files, details on how to use the generated parser in your programs, and details on the workings of the generated parsers. This manual can be found in the documentation directory. On Un*x systems this is usually /usr/share/doc/LLnextgen-0.5.5.