LLnextgen user manual

For version 0.5.5

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

31-12-2011

Contents

Chapter 1

Introduction

LLnextgen is a (partial) reimplementation of the LLgen ELL(1) [2] parser generator created by D. Grune and C.J.H. Jacobs[1] which is part of the Amsterdam Compiler Kit (ACK). As such, it creates C source-code for a text parsing engine from a description of the grammar. The parsers created use the LL(1) paradigm, with several extensions to allow for some ambiguities to be resolved without rewriting the grammar.

This manual is not an introduction to parsers or parsing paradigms. There are many books on parsing and compiler construction, for example [1].

Throughout this manual I have indicated where the behaviour of LLnextgen differs from LLgen with the ⏦ symbol in the margin. The manual page also provides an overview of the differences in behaviour of LLnextgen and LLgen. Furthermore, I have indicated several important issues, that are often overlooked with a ⚠ symbol in the margin.

1.1 Extent of reimplementation

LLnextgen implements the complete feature set of LLgen except for the extended user error-handling with the `%onerror` directive and the non-correcting error-recovery. The standard error-recovery mechanism is implemented.

The reason for not implementing the `%onerror` directive is because it is mostly a hook to allow research into different error-recovery mechanisms. This is not very useful for regular LLnextgen users.

The non-correcting error-recovery is not implemented because it is a lot of work and I think it is not an improvement over the default algorithm. Although it can produce fewer error messages, the location of the reported error can diverge from the location where the parser got stuck. My personal experience is that compilers that report errors at a different place then where the parser gets stuck can seriously hinder interpretation of the generated error-message by the compiler user.

[1] To add to the confusion, there exists or existed another program called LLgen, which is an LL(1) parser generator. It was created by Fischer and LeBlanc.

Chapter 2

Specifying grammars

2.1 Basic syntax

LLnextgen uses an EBNF-like syntax for specifying grammars. A grammar consists of rules, which in turn consist of elements. The elements in an LLnextgen grammar are terminals (or tokens), non-terminals (or rules), terms, actions and back-reference operators. The grammar file is also the place to specify several directives as well as providing code to be copied to the output (enclosed in braces).

Terminals can be either a character literal, specified as `'a'`, or an identifier. LLgen can handle the following character escapes: `'\b'`, `'\f'`, `'\n'`, `'\r'`, `'\t'`, `'\\'` `'\''` and octal character codes, for example `'\033'`. LLnextgen can also handle `'\a'`, `'\v'`, `'\?'`, `'\"'` and hexadecimal escape codes, for example `'\x1B'`. These all have the same semantics as in C, except for the hexadecimal escape codes. The hexadecimal escape codes only allow you to specify character literals up to 0xFF.

Non-terminals are specified by identifiers. Any non-terminal used in a rule, has to appear as a rule itself elsewhere in the grammar. As non-terminals are translated into C-functions, they can have arguments. The arguments can be passed in the normal way, that is, by writing a C-expression enclosed in parentheses. LLnextgen also supports return values. See Section 4.4 for details.

Terms are one or more elements enclosed in `[` and `]`. A `|` can be used to specify an alternation (or choice) between several alternatives.

Actions are pieces of C-code enclosed in braces. To determine the end of the C-code, LLnextgen tries to match the braces in the code and find the brace matching the opening brace of the action. This means that there is one restriction on the C-code: the number of opening braces must match the number of closing braces. Normally, C-code will satisfy this constraint, but if you are using `#define`s which contain braces LLnextgen's efforts to find matching braces may be thwarted.

A further restriction is that comments should not contain line continuations (a backslash followed by a newline) within the starting and ending delimiters (`/*`, `*/` and `//`). As this is something human programmers don't usually do, this is not a big restriction.

Back-reference operators are used to avoid code duplication. By specifying `...` in a term, all the elements preceding the term are included in its place. See Section 4.6 for more details.

Terminals, non-terminals and terms can all be followed by an optional repetition specification. The following list summarises the possible repetition specifiers:

- A number, specifying the exact number of times the element has to appear.
- A `+`, specifying the element has to appear 1 or more times.

- A `*`, specifying the element may appear 0 or more times.
- `+` or `*` followed by a number indicating the maximum number or times the element may appear.
- A `?`, specifying the element may appear once. This is a shorthand for `* 1`.
- The last item in a term followed by `+` or `*` operator can be followed by `..?` to indicate that this item is optional for the last repetition of the enclosing term. Each alternative of the term can contain an item followed by `..?`. A good example of its use is in specifying ISO C99 and C++ enums, where the last item in the enum can optionally be followed by a comma.

The LLnextgen grammar can be specified in the LLnextgen syntax. The example below is a simplified extract from the actual grammar file used to build LLnextgen. Of course, in the actual grammar the rules do have parameters and actions have been specified.

```
grammar : declaration * ;

declaration :
    C_DECL  /* Top level C-code */
|
    START IDENTIFIER ',' IDENTIFIER ';'
|
    /* Other declarations omitted for brevity */
|
    rule ';'
;

rule :
    identifier
    [   /* Return value type */
        '<'
        IDENTIFIER
        [
            '*'
        |
            IDENTIFIER
        ] *
        '>'
    ] ?
    C_EXPR ?  /* Parameters */
    C_DECL ?  /* Local variable declarations */
    ':'
    productions
;

productions :
    simpleproduction [ '|' simpleproduction ] *
;

simpleproduction :
    [
        DEFAULT
    |
        IF C_EXPR
    |
        /* Other directives omitted for brevity */
    ]
    [
        element
        repeats
```

```
    ] *
;

element :
    C_DECL  /* Action code */
|
    LITERAL
|
    IDENTIFIER  /* Can be both a terminal or a non-terminal */
    [   /* Return value renaming */
        '<'
        IDENTIFIER
        '>'
    ] ?
    C_EXPR ? /* Parameters */
|
    '['
    /* Optional directives omitted for brevity */
    productions
    ']'
|
    BACKREF /* The '...' operator */
;

repeats :
    /* No operator */
|
    [
        '*'
    |
        '+'
    ]
    NUMBER ?
|
    NUMBER
|
    DOTQMARK
;
```

Note the use of C-style comments. LLnextgen accepts both C-style comments and C++-style comments anywhere in the grammar. All capitalised words are terminals, although this is simply a convention. The `C_DECL` token represents a number of C declarations and statements, enclosed in braces. The `C_EXPR` token represents either a parameter list, as in `rule` and `element`, or an expression to be evaluated during parsing to resolve a conflict (as in `simpleproduction`). In both cases, the `C_EXPR` includes the enclosing parentheses.

The use of the rule `repeats` in `simpleproduction` could have used the `?` operator to obviate the need for an epsilon alternative in `repeats`. However, the LLnextgen grammar includes an action in the epsilon alternative (omitted in the example) which would not be executed if the `?` operator had been used.

2.2 Defining tokens

To use a token in the parser, it first has to be defined. To do this, one can use the `%token` directive, although as an extension to LLgen, one can also use `%label` for this (see Section 4.9). Multiple tokens can be declared by a single `%token` directive, for example:

```
%token IDENTIFIER, NUMBER, C_EXPR;
```

For quick development in the early stages, one can also use the –-token–pattern option which automatically

defines all the unknown identifiers that match the given pattern as tokens (see Section 4.10).

2.3 Conflicts

In LL(1) grammars, two kinds of conflicts can occur: FIRST/FIRST conflicts and FIRST/FOLLOW conflicts. LLgen names these alternation conflicts and repetition conflicts respectively (although repetition conflicts also cover cases that involve repetition operators). An alternation conflict occurs when two alternatives of a rule or a term can start with the same terminal. A repetition conflict can occur in two cases: when a term or rule has an empty alternative, and can be followed by a token that is also the start of one of the other alternatives, or when a repeating element with a variable repetition count (using `+` or `*`) can be followed by a token that is also the start of (an alternative of) the element.

One way to deal with conflicts is rewriting the grammar rules. For most cases this is the most practical way. However, in some cases it is possible to write an expression that determines which way to solve the conflict at run time. This is what the extended part of ELL(1) is about. Using the directive `%if` at the start of an alternative with a conflict, followed by a C-expression in parentheses, an alternation conflict can be resolved. For the common expressions (1) and (0), the directives `%prefer` and `%avoid` have been defined. Using these instead of `%if (1)` and `%if (0)` will produce faster code. Note that these directives cannot be used on the last conflicting alternative as there always has to be a fall-back alternative for each conflicting token.

An example of an alternation conflict is the following grammar:

```
 1: %start parser, starting_rule;
 2: %token A;
 3:
 4: starting_rule :
 5:     A
 6: |
 7:     rule
 8: ;
 9:
10: rule :
11:     'a'?
12:     A
13: ;
```

If LLnextgen is run on the preceding grammar, with the –-verbose option, it will output the following:

```
<stdin>:6: error: Alternation conflict with alternative at <stdin>:5 in
starting_rule
Trace for the conflicting tokens from alternative on line 5:
  A [ line 5 ]
Trace for the conflicting tokens from alternative on line 6:
  rule [ line 7 ] ->
    A [ line 12 ]
```

The trace of the first alternative is straightforward. It specifies that on line 5 the token `A` is part of the first set of the alternative. The trace for the second alternative specifies that the conflicting token is in `rule` which is called from line 7. LLnextgen then goes on to show where in `rule` the offending token is mentioned, in this case on line 12.

For repetition conflicts one can use the `%while` directive at the start of a repeating term, again followed by a C-expression in parentheses. If the expression evaluates to something other than zero, the repetition will be continued. If the expression evaluates to zero, parsing will continue with the grammar following the repetition[1].

In the example below there is a repetition conflict on line 15. LLnextgen cannot decide whether to match the optional `'a'` at line 15, as the rule `inner` may also be followed by a `'a'` (through the call to `outer` on line 4).

[1] Due to a bug in LLgen, `%while` alternatives could not be used in `+` repetitions. LLnextgen on the other hand does allow these.

```
 1: %start parser, outest;
 2:
 3: outest :
 4:     [ outer ] +
 5:     'a'
 6: ;
 7:
 8: outer :
 9:     'c'
10:     inner
11: ;
12:
13: inner :
14:     'b'
15:     'a'?
16: ;
```

If LLnextgen is run on the preceding grammar, with the –-verbose option, it will output the following:

```
<stdin>:15: error: Repetition conflict in inner
Trace for the conflicting tokens from the first set:
  'a' [ line 15 ]
Trace for the conflicting tokens from the follow set:
  <- inner from outer [ line 10 ]
    <- outer from outest [ line 4 ]
      'a' [ line 5 ]
```

In the trace, the left arrows (`<-`) indicate that the follow set of the rule being traced (the first rule mentioned) is at least partly determined by a call to that rule at the specified location. In this case, the follow set of the rule `inner` is determined by a call in rule `outer` on line 10. As this is the last part of `outer`, the follow set of `inner` is further determined by the location where `outer` is called. The next line of the trace therefore specifies that there are tokens in `inner`'s follow set which come from the call to `outer` on line 4. This call to `outer` can be followed by the `'a'` on line 5, which is the source of the conflict.

An example of using a `%while` directive to solve this conflict would be the following: if, for example, the `'a'` on line 15 must only be matched the first 10 times, one could change the rule `inner` into:

```
inner { static int count = 0; } :
    'b'
    [
        %while (++count <= 10)
        'a'
    ] ?
;
```

Note however, that using the `static` variable this way means that you can only call the parser once. Using a global variable which is reset at some point will allow for calling the parser multiple times.

To aid expression-writing, the `%first` directive can be used to declare macro's that evaluate to one if a rule can start with a given token. For example, declaring `%first fset, rule;` declares a macro named `fset` that takes a single argument, the number of a token. If that token can start the rule `rule`, `fset` evaluates to one.

Chapter 3

Interfaces

This chapter details the interfaces expected and provided by the parsers generated by LLnextgen.

3.1 Name prefixes

All symbols (functions and variables) generated by LLnextgen are by default prefixed with LL. To facilitate multiple parsers in one program, LLnextgen can be instructed to use a different prefix for all symbols with external linkage. This is accomplished using `#define`s, so that within the output C file the symbols can still be used with their `LL` prefix. To instruct LLnextgen to use a different prefix, use a `%prefix` directive like for example `%prefix PF;`.

NOTE: it is inadvisable to create symbols using the LL prefix, or any prefix specified with `%prefix`. Doing so can cause name-clashes at both compile and link time.

3.2 Generated files

LLnextgen generates two files by default: a .c file and a .h file. The base name of the files is the name of the first input file with an optional trailing .g extension removed, or the name specified by the –-base–name option. The default extensions can be overriden by using the –-extensions option. This is different from LLgen, which generates three files by default: a .c file for each input file, a file named Lpars.c and a file named Lpars.h. If a `%prefix` directive has been specified in the grammar, the latter two files would have the prefix in place of the capital L. The LLgen behaviour can be obtained by specifying the –-llgen–output–style option.

The header file contains `#defined` constants for all the tokens defined through `%token` and `%label` directives, as well as for the symbols `EOFILE`, `LL_MAXTOKNO`, `LL_MISSINGEOF` and `LL_DELETE`. The token `#define`s are enclosed in a conditional compilation block. If the symbol `LL_NOTOKENS` is `#defined`, the tokens will not be available.

NOTE: the guard symbol is always named `LL_NOTOKENS` regardless of any `%prefix` directives and the symbols `LL_MAXTOKNO`, `LL_MISSINGEOF` and `LL_DELETE` are excluded from the conditional compilation.

Finally, the header file also contains prototypes for the parser itself, and if applicable also for `LLreissue` (see Section 3.3) and `LLabort` (see Section 4.5).

3.3 Lexical analyser

LLnextgen needs to be provided with a lexical-analyser routine. The lexical analyser is expected to return an `int`; the token number. The token numbers 1 through 255 have been reserved for character literals. This includes the standard ASCII character set. The tokens defined through `%token` and `%label` directives also have constants `#defined` for them in the generate header file. Token number 0 is normally reserved

for signalling the End-Of-File condition, but using the option –-no–eof–zero, this token can be used for the nul character. However, you need to ensure that your lexical analyser returns a proper End-Of-File marker. Therefore, the token number -1 is also reserved for signalling the End-Of-File condition, as is the `#define`'d constant `EOFILE` in the generated header file. For flex based lexical analysers, you also have to manually specify that you want to return either of these values instead of 0 to signal the End-Of-File condition. For example by including the pattern:

```
<*><<EOF>>          { return EOFILE; }
```

in your lexical analyser.

It is also possible to use the EOFILE token in your grammar. However, not all lexical analysers can be called again without resetting, after having returned an End-Of-File condition. In particular, flex scanners are explicitly specified to give undefined results in this case. Therefore LLnextgen will generate a warning if you do use EOFILE in your parser. This warning can be disabled with –-suppress-warnings=eofile, but you must make sure that your lexical analyser will be reset, or will keep returning the End-Of-File condition on repeated calls.

Another short warning is in order at this time: when returning a single character as a token, make sure you return a value greater than zero. By default characters are signed in C, so simply returning a `char` variable in the lexical analyser can cause problems. For a (f)lex based lexical analyser the best way to return a single character is: `return * (unsigned char *) yytext;`.

The name of the lexical-analyser routine defaults to yylex, to facilitate easy integration with (f)lex generated analysers. To specify a different name for this routine, supply a `%lexical` directive in your grammar. For example, `%lexical scanner;` would indicate that the analyser to be used is named `scanner`.

LLnextgen requires lexical analysers to return the same token returned previously after it inserts a token during error recovery. Most lexical analysers do not support this kind of unput action, so a wrapper has to be written for the lexical analysers. As this usually leads to the same code for each parser, LLnextgen can generate a default wrapper by specifying the –-generate–lexer–wrapper. This default wrapper can also be dumped on standard output by using –-dump–lexer–wrapper. From version 0.5.1, LLnextgen will issue a warning if the –-generate–lexer–wrapper option is not specified. If you do not want the automatically generated wrapper, specify –-generate–lexer–wrapper=no in your options.

To help write wrappers for lexical analysers the variable `LLreissue` is set to the token that needs to be reissued, or `LL_NEW_TOKEN` if no reissue is requested (unless the option –-no–llreissue is specified). The lexer wrapper is expected to reset `LLreissue` to `LL_NEW_TOKEN` after reissuing the previous token.

NOTE: versions of LLnextgen before 0.3.0 use a different and incompatible convention for the value of `LLreissue`. To distinguish between these versions, use the C macro `LL_VERSION` (see Section 4.1). Incompatible older versions of LLnextgen do not define this macro. However, if you use –-generate-lexer-wrapper this difference does not concern you as it is hidden by the generated code.

The result of the last call to the lexical analyser is stored in `LLsymb`. The value of `LLsymb` is only valid in `%while` and `%if` directives *preceding* a token matching, and in actions *following* a token matching. In the arguments of rules named after the token matching, `LLsymb` contains a different value. `LLsymb` can also be used in `LLmessage` to determine the identity of the token skipped, or the token in front of which a different token will be inserted (see Section 3.5).

3.4 Parser routine

The generated parser must be given a name and a starting rule with the `%start` directive. The syntax is `%start` *`parser_name`*`,` *`starting_rule`*`;`. The generated parser will then have the following prototype:

```
void parser_name(void);
```

However, there are three cases where the prototype changes: when the option –-abort is used, the return type changes to `int` (see Section 4.5). When the option –-thread–safe is used the parser will take an argument (see Section 4.8). Finally, when the *starting_rule* has a return value, the parser will also

take an argument (see Section 4.4). The prototype will be added to the header file, unless the option ––no–prototypes–header is used.

3.4.1 Multiple parsers in one grammar

It is possible to specify multiple parser in the same grammar file. This can be useful if the parsers have many common rules, or share a single lexical analyser. These parsers will share common data structures. If the parsers are not called from one another this causes no problem. For the case where the parser do call one another, either the option ––reentrant or the option ––thread-safe needs to be specified (See also Section 4.7 and Section 4.8). LLnextgen will issue a warning if you don't specify one of these option but do use multiple parsers in one grammar. To suppress this warning use ––suppress-warnings=multiple-parser.

3.5 Error handling

When the generated parser encounters an error in the input, it tries to find a sequence of token deletions and insertions such that the token encountered can be used as part of the grammar. To decide which tokens to discard, the parser keeps track of the set of tokens which will always be matched by continuing after the error correction. To determine this set, LLnextgen uses so called default choices for each alternation. A default choice is the alternative that is chosen when trying to recover from an error. By default, it is the first alternative that needs the minimum number of tokens to complete. This is a slight deviation from the LLgen way of choosing a default choice, as LLgen also takes the complexity of the alternatives into account. There is also a difference in the handling of the `%avoid` directive. LLnextgen deems all alternatives marked with `%avoid` equal and simply chooses the first (if they are the shortest alternatives that is), while LLgen chooses the last.

These differences are not very large, and can be circumvented by specifying a `%default` directive on the alternative of choice. It should be noted that the `%default` directive is also an effective means of error recovery in the actions. It is usually easiest to write an action without regard to the validity of the token text, especially when the token text is expected to be something other than a keyword or operator. To direct the parser to use a different action to handle these situations, one might use the following construction:

```
[
    IDENTIFIER { /* add to symbol table */ }
|
    NUMBER     { /* extract value from text */ }
|
    %default MISSING_EXPRESSION /* No action */
]
```

For terms with a variable repetition count (i.e. terms followed by `+`, `*` or `?`), the default is to assume a minimal number of repetitions. If however it is desirable to make the parser to go into the repetition if one of the tokens of the default choice within the repetition appears, a `%persistent` directive can be added at the start of the term.

The parser calls the routine LLmessage, which has to be provided by the parser writer, to indicate either a deletion or an insertion of a token. LLmessage takes one `int` parameter, which can have the one of the following values:

`LL_DELETE` to indicate that the parser is about to discard the token with token number `LLsymb`.

`LL_MISSINGEOF` to indicate that the parser expected the end of input, and is about to discard the token with token number `LLsymb`.

Any other value to indicate the parser is about to insert a token with that number before the token with token number `LLsymb`.

Note that LLgen uses the fixed values 0 for `LL_DELETE` and -1 for `LL_MISSINGEOF`. Because LLnextgen uses 0 as a regular token number when the option ––no–eof–zero is used, the value 0 would

have two meanings. Therefore the use of the hard coded values 0 and -1 is deprecated, and all new parsers should use `LL_DELETE` and `LL_MISSINGEOF` instead.

To make development quicker, a default LLmessage routine can be generated using the option ––generate–llmessage. This routine can also be dumped to standard output by specifying the option ––dump–llmessage on the command line. The default version can then be used as a starting point for a more elaborate message printing routine.

Chapter 4

Other features

This chapter describes several features LLnextgen provides over the standard LLgen feature set. These have been created to make development easier.

4.1 Version macro

To distinguish different interface versions, LLnextgen defines the C macro `LL_VERSION` which is affected by `%prefix`. It has been defined since version 0.3.0. The value of the macro is the version number encoded as a hexadecimal number, with two digits per version number position. The number for version 0.3.0 is therefore 0x000300. Currently, the only interface for which distinguishing versions is necessary is the `LLreissue` variable (see Section 3.3).

4.2 Including files

LLnextgen has a file inclusion mechanism, similar to the `#include` mechanism in C. To include another grammar file use: `%include "`*`filename`*`";`. The filename may include C-style escaped characters. LLnextgen tries to prevent you from including files recursively, and will abort with an error if it detects this.

4.2.1 Dependencies

The include mechanism also introduces a dependency situation. This requires proper handling in Makefiles. To help developers, LLnextgen can provide dependency information for its input files. Using the option –-depend will print a line with the names of the files that will be created, followed by a semicolon, followed by all the input files that will be used to create the output files. Several modifiers exist to change the output (see Appendix A).

A problem that already existed with LLgen is that to find out which header files the generated code needs, one needs to generate the code. However, for dependency generation it is undesirable to already generate the parser code. Therefore LLnextgen adds an option to simply dump all the top-level C-code (-–depend–cpp). Piping this through the C preprocessor allows dependency generation to proceed without generating the actual parser. For example:

```
LLnextgen --depend-cpp grammar.g | gcc -E -MM -MP -MG -MT 'grammar.o' -
```

can be used to generate dependency information for `grammar.o`, using gcc. This does of course require that the input is syntactically correct.

4.3 Specifying options in the grammar

As options for LLnextgen can be specific to a grammar, it is logical to allow grammar writers to specify the options in the grammar as well. This can be done with a `%options` directive. The `%options` directive must be followed by a double quoted string with options. Only long options can be specified and the leading dashes must be omitted. The string is processed for C-style escaped characters.

4.4 Return values

From LLnextgen version 0.4.0, rules can have return values. To use return values, a rule needs to have a return type. This can be specified by naming the desired type between '`<`' and '`>`' *after* the name of the rule. The name of the type can consist only of identifiers and '`*`' operators. To use a type that contains other characters (for example the '`<`' character for C++ templates) you need to use a `typedef`.

The return value of a rule is the last value assigned to the generated local variable `LLretval`. `LLretval` will by default be filled with 0 bytes, unless the option ––no–init–llretval is used.

For each rule that is called, a local variable is created that will contain the value returned by that rule. This variable can be freely used in your C code. By default, that variable will have the name of the rule that returned the value. However, by using an identifier enclosed in '`<`' and '`>`' the variable can be given a name of your choice. Two rules can also use the same variable to return their value, if they have the same return type. The calculator example in Section 5.1 uses return values.

If the starting rule for a parser has a return value, the prototype for the parser is changed. The parser will take a single argument, which is a pointer to a variable where the return value of the rule should be stored. It's type is therefore a pointer to the type named as the return type of the rule. For thread safe parsers, the return value argument is the second argument.

LLnextgen will also try to warn you if a rule returns a value, but the returned value is ignored. However, because LLnextgen does not look inside code segments, it assumes that a value returned before a code segment is used in the subsequent code. The same holds for arguments passed to a subsequent rule. If you do not intend to use the value returned from the rule, you can rename its return value to `LLdiscard`. LLnextgen will not warn about return values assigned to `LLdiscard`, and will not create a local variable to hold the returned value.

4.5 LLabort

It is not always desirable to continue parsing after an error. To accommodate this, LLnextgen can be instructed to generate a routine called LLabort. This routine has to be passed one integer with a value other than 0. As mentioned in Section 3.4, this option changes the prototype of the parser routine such that it returns an `int`. The value returned is 0 if the parser completed normally, and the value passed to LLabort otherwise.

4.6 Back-reference operator

The back-reference operator (`...`) can be used in a term to prevent code duplication. All the elements preceding the term are copied in its place. Note that this also means you cannot use labels (for use with `goto`) in referenced actions, and static variables will be duplicated.

This construct is most usefull in the situations like comma-separated parameter lists, which are usually specified as follows:

```
parameterList :
    type
    IDENTIFIER
    { /* code */ }
    [
        ','
        type
```

```
        IDENTIFIER
        { /* code */ }
    ] *
;
```

where type is a non-terminal and IDENTIFIER a terminal. With the back-reference operator this can be specified in the following, shorter, way:

```
parameterList :
    type
    IDENTIFIER
    { /* code */ }
    [
        ','
        ...
    ] *
;
```

Note that only the elements in the enclosing term are copied. For example, in:

```
rule :
    'a'
    [
        'b'
        [
            ','
            ...
        ]*
    ]
;
```

only the literal `'b'` is copied, and not the literal `'a'` because it is outside the enclosing term.

4.7 Reentrant parsers

The parsers generated are not reentrant by default, contrary to LLgen parsers. To make the parsers reentrant (**NOTE**: not thread-safe!), use the ––reentrant option. This makes it possible for the parser to call itself. This is different from running two parsers simultaneously in different threads. See Section 4.8 for information on thread-safe parsers.

Calling the parser will change the state of the lexical analyser, which the currently running parser relies on. It is therefore important to use a reentrant lexical analyser as well, when using reentrant parsers. Flex provides these from versions after 2.5.4a (using `%option reentrant` or ––reentrant). Older versions of flex do provide a way to switch between buffers, but this method fails to save the contents of`yytext` and is therefore unsuitable for most cases. It is possible to save yytext yourself and thereby still use the buffer switching mechanism flex provides.

Reentrant parsers are a way to implement file inclusion when specific tokens are expected after the include command. For example, in LLnextgen a semicolon (;) is expected after the string containing the file name. It is of course also possible to incorporate the recognition of include statements completely in the lexical analyser. However, that would cause a substantial amount of work if for example comments are to be allowed between the tokens as well. Below is the (simplified) code from the LLnextgen grammar:

```
INCLUDE
[
    STRING
    { token = newToken(); }
|
    %default
    MISSING_STRING /* token is NULL by default */
```

```
]
';'
{
    if (token != NULL) {
        if (openInclude(token))
            parser();
        freeToken(token);
    }
}
```

When the end of an include file is reached, the lexical analyser needs to return the end-of-file token and switch back to the previous file. However, returning the end-of-file token may cause error messages from the parser. To provide proper indication of where (what line in which file) the error is, the line number and file name information should not be reset until the next token is to be retrieved from the lexical analyser. When using flex this can be achieved in the following way:

```
int switchBack = 0;

int yywrap(void) {
    ...
    switchBack = 1;
    ...
}

int lexerWrapper(void) {
    if (LLreissue == LL_NEW_TOKEN) {
        if (switchBack) {
            /* switch back to previous lexer state */
        }
        ...
    } else {
        ...
    }
}
```

As you can see, this requires a hand crafted lexer wrapper.

4.8 Thread-safe parsers

LLnextgen can also generate thread-safe parsers. This is required when multiple instances of the same parser are to run in parallel. To make LLnextgen generate a thread-safe parser, the option –-thread–safe needs to be specified. This will define the macro `LL_THREAD_SAFE` (which is affected by `%prefix`) and change the interface to several functions. The sections below detail the changes to the interface with respect to the standard interface. For an example, see Chapter 5.

4.8.1 Parser routines

Parser routines for thread-safe parsers take an argument. This argument is meant for passing data to and from the parser. The argument passed is available to all functions generated as part of the parser through the macro `LLdata`. By default this argument is of type `void *`. To change the type of the argument, the `%datatype` directive can be used. Its syntax is as follows:

```
%datatype "type" [, "header file"];
```

The first argument is the type of the argument to the parser. If the type is not a standard C type, inclusion of a header file with the type declaration is required. The required header file can be specified with the second argument. By default the header file is assumed to be a local header file. However, if the string is enclosed in '<' and '>', the header is assumed to be a system header file. If `%datatype` is used in a non-thread-safe parser, LLnextgen will issue a warning which can be suppressed with –-suppress–warnings=datatype.

4.8.2 LLmessage and lexical analyser

The `LLmessage` function, as well the lexical analyser both have an extra argument named `LLthis`. It contains the parser state and is of type `struct LLthis *`. It also contains a member named `LLdata_` which contains the user data. The macro `LLdata` expands to `LLthis->LLdata_` to ease access. It is intended that the user data contained in this member also contains the state for the lexical analyser. Note that the name of the type and the name of the macro are changed by a `%prefix` directive, but the name of the argument is not.

For LLmessage, the new signature is:

```
void LLmessage(struct LLthis *LLthis, int token);
```

4.8.3 LLabort

If the `LLabort` function is enabled with the –-abort option, its signature is changed into:

```
void LLabort(struct LLthis *LLthis, int retval);
```

4.9 Symbol tables

When printing error messages, it is often desirable to have a string associated with a token number. To accommodate this, LLnextgen can create a symbol table (using the option –-generate–symbol–table).

By default all tokens that have been created with `%token` have as associated string the token name itself. For example, if `%token IDENTIFIER;` appears somewhere in the grammar, the string associated with the token number for IDENTIFIER would be ”IDENTIFIER”. The default for the character literals is the table defined in the LLnextgen source code. For the characters up to and including space and for character 127, it is the name of the control character enclosed in <>. For characters between space (32) and 127 it is the character itself enclosed in single quotes (`'`), and for all other characters it is the hexadecimal C-style escape code enclosed in single quotes.

All these defaults can be overridden by the `%label` directive. Its syntax is:

```
%label token, string;
```

`token` can be both a character literal or an token identifier. String is output unprocessed to the output file. A token identifier does not have to be declared by a `%token` directive, unless the option –-no–allow–label–create has been specified.

To use the symbol table, use the function `LLgetSymbol`. It takes a token number as only argument, and returns a pointer to a string constant, or NULL if the token number is invalid.

4.9.1 Symbol tables and gettext

For internationalised programs, the strings returned by `LLgetSymbol` may need to be translated. LLnextgen provides the –-gettext option, which will ensure that all symbol names specified by a `%label` directive are enclosed in a macro call. The macro will expand to the string itself. This way, one can use **xgettext** to extract the strings to be translated. The default macro name 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. The –-gettext option takes optional names for the macro and guard, separated by a comma, as arguments.

4.10 Automatic token declarations

Note: the following options are 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.

In the early stages of development it can be a nuisance to have to define all the tokens used in the grammar, simply to test for conflicts. To mitigate this problem LLnextgen provides the –-token–pattern option. The argument to the –-token–pattern option is a regular expression that is used to test if an unknown identifier is meant to be a token, or maybe is a misspelled rule name.

When the grammar has stabilised, the –-dump–tokens can be used to generate a list of token declarations for the identified tokens. The default is to output a single `%token` directive which includes all token definitions. The –-dump–tokens takes a single optional argument which modifies the way the declarations are printed. The *separate* modifier makes LLnextgen output a separate `%token` directive for each identifier, while the *labels* modifier makes LLnextgen output a `%label` directive for each identifier. The text for the label is the name of the identifier. If the *labels* modifier is used in combination with the –-lowercase–symbols option, the text for the label will contain only lowercase characters.

For example, given the following grammar:

```
rule:
    TOKEN
    IDENTIFIER
;
```

using the options –-token–pattern=^[A-Z]+$ and –-dump–tokens will result in the output:

```
%token TOKEN, IDENTIFIER;
```

If instead –-token–pattern=^[A-Z]+$ –-dump–tokens=labels –-lowercase–symbols is used, the output will be:

```
%label TOKEN, "token";
%label IDENTIFIER, "identifier";
```

Without –-dump-tokens the grammar will be accepted as if the above declarations were included in the grammar.

4.11 %top C code

Sometimes it is necessary to include some definitions before any other code in the generated parser file. To facilitate this, a single section of C code may be marked as top code, by prefixing it with %top.

Chapter 5

Examples

This chapter contains two examples. The first is a very simple calculator, which shows basic LLnextgen use and a sophisticated use of `%while`. The second is an example of the thread-safe parser interface.

Warning: when copying the text below, make sure that you remove any page numbers and take care to ensure all characters in your text file are ASCII characters and not UTF-8 or other characters. Another option is to use the example files from the documentation directory.

5.1 Calculator

The file below shows a very simple calculator. It uses only integer numbers, and can add (`+`), subtract (`-`), divide (`/`), multiply (`*`), take the modulo (`&`), and calculate powers (`^`).

```
%start calculator, input;
%label NUM, "number";
%options "generate-lexer-wrapper generate-llmessage generate-symbol-table";
%lexical lexer;

{
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <math.h>

static int value;

enum states {
    START,
    NUMBER
};

int lexer(void) {
    enum states state = START;
    int c;

    value = 0;
    while ((c = getchar()) != EOF) {
        switch (state) {
            case START:
                if (isspace(c) && c != '\n') {
                    /* Skip white space, except for newlines. */
                    continue;
                }   else if (isdigit(c)) {
```

```
                    /* Digits mean a number! */
                    state = NUMBER;
                    value = c - '0';
                    break;
                }
                /* Simply return all other characters and let the
                   parser error handling sort it out if necessary. */
                return c;
            case NUMBER:
                /* Read all digits and push back the non-digit, so
                   we can reread that the next time. */
                if (!isdigit(c)) {
                    ungetc(c, stdin);
                    return NUM;
                }
                value = value * 10 + (c - '0');
                break;
        }
    }
    /* We're done. */
    return EOFILE;
}

/* Simple main routine to fire up the calculator. */
int main(int argc, char *argv[]) {
    printf("LLnextgen integer-calculator example. Press ^C or ^D to end.\n");
    calculator();
    return 0;
}

/* Define the operator priorities. A table would have been possible
   as well, but this is just as clear and requires less memory. */
int getPriority(int operator) {
    switch(operator) {
        case '-':
        case '+':
            return 0;
        case '/':
        case '%':
            return 1;
        case '*':
            return 2;
        case '^':
            return 3;
    }
    /* This should never happen. */
    abort();
}

}

input :
    '\n' *        /* Empty lines should be skipped. */
    [
        expression(0)
        {
            printf("Answer: %d\n", expression);
        }
```

```
        '\n' +    /* Empty lines should be skipped. */
    ] *
;

expression<int>(int priority) :
    /* Expressions are factors (numbers, negated expressions and
       expressions between parentheses) followed by operators,
       followed by expressions with higher priority. */

    factor<LLretval>
        /* By renaming the return value of expression to LLretval, we
           immediately set the return value of this rule. */
    [
        %while (getPriority(LLsymb) >= priority)
        /* The %while directive says to keep accumulating operators
           as long as they have equal or higher priority. */

        '-'
        expression<intermediate>(getPriority('-') + 1)
        /* The getPriority() + 1 means that '-' is left associative.
           If it needs to be right associative, this needs to be
           getPriority().
           Also note the explicit use of '-' instead of LLsymb. This
           is necessary as LLsymb has changed after matching '-'. */
        {
            LLretval -= intermediate;
        }
    |
        '+'
        expression<intermediate>(getPriority('+') + 1)
        {
            LLretval += intermediate;
        }
    |
        '*'
        expression<intermediate>(getPriority('*') + 1)
        {
            LLretval *= intermediate;
        }
    |
        '/'
        expression<intermediate>(getPriority('/') + 1)
        {
            LLretval /= intermediate;
        }
    |
        '%'
        expression<intermediate>(getPriority('%') + 1)
        {
            LLretval %= intermediate;
        }
    |
        '^'
        expression<intermediate>(getPriority('^') + 1)
        {
            LLretval = (int) pow(LLretval, intermediate);
        }
    ] * /* Note: an expression can also be just a number or parenthesised
```

```
            expression, so there can also be 0 operators. Hence the *. */
;

factor<int> :
    '('
    expression<LLretval>(0)
    ')'
|
    '-' expression(1)
    {
        LLretval = - expression;
    }
|
    NUM
    {
        LLretval = value; /* value is set by the lexical analyser. */
    }
;
```

The main thing to note is the use of `%while` to achieve operator precedence. Each time an operator is matched, `expression` is called recursively to match a part of the input containing only operators with greater precedence. After `expression` is done with matching the subexpression, more operators are matched at the current level or higher. This can be used in compilers as well.

5.2 Thread-safe parser

The parser below does not do anything particularly useful. It is simply meant to show the interface for thread-safe parsers. The parser uses the following header file:

```
#ifndef DATA_H
#define DATA_H

struct data {
    char *string;
    int index, dontStop;
};

#endif
```

And this is the parser:

```
%options "thread-safe abort generate-lexer-wrapper generate-symbol-table";
%datatype "struct data *", "data.h";
%start parser, rule;
%lexical lexer;

rule :
    'a'+
;

{
#include <stdio.h>
#include <stdlib.h>

int lexer(struct LLthis *LLthis) {
    return LLdata->string[LLdata->index++];
}
```

```
void LLmessage(struct LLthis *LLthis, int LLtoken) {
    switch (LLtoken) {
        case LL_MISSINGEOF:
            fprintf(stderr, "Expected %s, found %s.\n",
                LLgetSymbol(EOFILE), LLgetSymbol(LLsymb));
            break;
        case LL_DELETE:
            fprintf(stderr, "Unexpected %s.\n",
                LLgetSymbol(LLsymb));
            break;
        default:
            fprintf(stderr, "Expected %s, found %s.\n",
                LLgetSymbol(LLtoken), LLgetSymbol(LLsymb));
            break;
    }

    if (!LLdata->dontStop)
        LLabort(LLthis, 1);
}

int main(int argc, char *argv[]) {
    struct data data;
    int i;

    for (i = 1; i < argc; i++) {
        data.string = argv[i];
        data.index = 0;
        /* Don't stop for odd numbered arguments. */
        data.dontStop = i & 1;
        if (parser(&data) == 1) {
            printf("Failed at argument %i\n", i);
            exit(EXIT_FAILURE);
        }
    }
    exit(EXIT_SUCCESS);
}

}
```

Chapter 6

Contact

6.1 Reporting 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. Bug reports can be sent to <`llnextgen@ghalkes.nl`>.

6.2 Let me know

If you have suggestions for improving LLnextgen, write me an e-mail at <`llnextgen@ghalkes.nl`>.

If you use LLnextgen in one of your programs, please let me know. Send me an e-mail at the aforementioned address, preferably with a link to your project and whether you would like to be mentioned on the LLnextgen webpage.

Bibliography

[1] Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs, and Koen G. Langendoen. *Modern Compiler Design*. John Wiley & Sons, Ltd., 2000.

[2] Ceriel J. H. Jacobs. Some topics in parser generation. Technical Report IR-105, Department of Computer Science, Vrije Universiteit, Amsterdam, 1995. `http://www.cs.vu.nl/~ceriel/LLgen.html`.

Appendix A

Manual page

NAME

LLnextgen – an Extended-LL(1) parser generator

SYNOPSIS

LLnextgen [*OPTIONS*] [*FILES*]

DESCRIPTION

LLnextgen is a (partial) reimplementation 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.

–E*num*, –-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 specifiy 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 specifiy 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–eof–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.

–h*num* 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

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.