DFASTAR 4.0

User's Manual

April 2013

Acknowledgments

Since many of the ideas used in the development of DFASTAR were not entirely the author's invention it seems appropriate to give credit to those who contributed to this effort whether they knew it or not. The following publications played a key role in the development of DFASTAR.

1. "**Principles of Compiler Design",**
Book from Addison Wesley, 1977, by Aho and Ullman.

2. **"Compiler Construction",**
Book from Springer Verlag, 1985, by Waite and Goos.

3. **"Efficient Computation of LALR(1) Look-Ahead Sets"**
Paper from TOPLAS Oct 1982, by DeRemer and Pennello.

4. **"Optimization of Parser Tables for Portable Compilers",**
Paper from TOPLAS Oct 1984, by Dencker, Durre and Heuft.

5. **"A Human-Engineered Variant of BNF",**
Paper from Sigplan Notices 1980, by Henry Ledgard.

And finally, I wish to acknowledge Almighty God our Creator. Any talents that I have manifested in the creation of this product, have come from Him.

Table of Contents

Chapter 1.
Installation

The DFASTAR software is supplied with LRSTAR and resides in the bin directory, which is one of the following, the first level inside the root, LRSTAR.

bin
doc
grm
project
skl

The bin folder.

Inside the 'bin' folder you should have:

dfastar.exe	(the DFASTAR program)
lrstar.exe	(the LRSTAR progoram)

Chapter 2.
Command-line syntax.

DFASTAR is a command-line (or console-mode) program. The command-line syntax for proper usage is as follows:

```
dfastar <grammar> [<skeleton> <output>]... [/<option>]...
```

Where 'dfastar' is the program name (the 'dfastar.exe' file found in the 'bin' folder). The projects have been set up to access this file so that you do not have to redefine the Path in Windows (DOS).

Where '<grammar>' is the input grammar file, such as 'calc.lgr' or just 'calc'.

Where '[<skeleton> <output>]...' means zero or more skeleton and output files specified in pairs.

Where '<skeleton>' is an input skeleton file, such as: 'lexer.cpp.skl'.

Where '<output>' is an output file, such as: 'lexer.cpp'.

Where '[/<option>]...' means zero or more program options.

Where '<option>' is a one of the program options:

Here are some examples:

```
dfastar ansic.lgr
dfastar ansic lexer.cpp.skl lexer.cpp /s /tm /m /v
dfastar ansic lexer.cpp.skl lexer.cpp lexer.h.skl lexer.h /v /ts /m
```

Here is a typical **make.bat** file for building a parser (on Windows systems):

```
..\..\bin\lrstar calc.grm                ^
..\..\skl\parser.h.skl   calc_parser.h   ^
..\..\skl\parser.cpp.skl calc_parser.cpp ^
/k /v /s /exp /ct /d

if errorlevel == 1 goto :end

..\..\bin\dfastar calc.lgr             ^
..\..\skl\lexer.h.skl   calc_lexer.h   ^
..\..\skl\lexer.cpp.skl calc_lexer.cpp ^
/line /v /m

:end
```

Lastly, if you just type '**dfastar**' without any arguments then you will get a display showing the program options, as follows:

```
DFASTAR 4.0.004 Copyright 2013 Compilerware.
|
|   DFA Lexer Generator
|   dfastar <grammar> [<skeleton> <output>]... [/<option>...]
|
|   OPTION  DEFAULT  DESCRIPTION
|   a           0    Analyze the grammar only
|   col         0    Column number counting in lexer
|   d           0    Debug option for generated lexer
|   e          10    Error count maximum for grammar errors
|   g           0    Grammar listing
|   i           0    Case insensitive ('a'='A')
|   k           1    Keyword recognition
|   ki          1    Keyword and <identifier> recognition
|   line        0    Line number counting in lexer
|   m           0    Minimize lexer-table size
|   q           0    Quiet mode, minimal screen display
|   s           0    State machine listing
|   tab         0    Tabs setting in grammar
|   ts          0    Table-driven small lexers
|   tm          0    Table-driven medium lexers, 10% faster
|   v           0    Verbose mode, list usage data
|   w           0    Warning messages listing
|   wait        0    Wait on key press to end program
|
0 min 0.000 sec, 0.000 MB, 0 warnings, 0 errors.
.
```

Chapter 3.
Program Options

This chapter contains a more in depth explanation of the DFASTAR program options. Here is the list of options.

Option	Description	Default Value
a	Analyze the grammar only.	0
col	Column number counting in the lexer.	0
d	put level 1 Debug code in parser.	0
e=?	Error count before quitting.	10
g	Grammar listing output.	0
i	Case insensitive treatment of input.	0
k	Keyword recognition.	1
ki	Keyword and <identifier> recognition.	1
line	Line counting in lexer.	0
m	Minimize parser table size.	0
n	Number of characters in set (128, 256).	256
q	Quiet mode, minimal screen display.	0
s	show State machine listing.	0
tab=?	Tab setting in grammar file.	0
ts	Table-driven small lexers.	1
tm	Table=driven medium lexers.	0
v	Verbose mode, show more information.	0
w	show Warning messages in **.log.txt** file	0
wait	Wait for key press, before exiting program.	0

'a' option: Analyze the grammar only.
This is useful if you are working with a huge grammar and it takes a long time to generate the parser tables and all you want to do is to get a list of conflicts in your grammar. Most people will never need this option because DFASTAR generates parser tables quickly, except for very large grammars (more than 10,000 rules). The default is 'a=0' (do the complete parser generation process).

'col' option: Column number counting in lexer.
The 'col' option tells DFASTAR to turn on the @optn_col skeleton code setting it to one. The generated lexer can then include code to compute the column number in the input stream.

'd' option: Debug information .
The 'd' option tells DFASTAR to turn on the @optn_debug skeleton code setting it to one. Then the lexer which can include code that will display extra information while running useful for debugging.

'e' option: Error count limit.
The 'e=?' option tells DFASTAR how many errors to show before quitting. However, some errors will stop the DFASTAR program before listing any other errors. The default number of errors to display is 10.

'g' option: Create a grammar listing file.
The 'g' option tells DFASTAR to generate a grammar file: "?.lgr.grammar.txt".

'i' option: Case insensitive treatment of input characters.
The 'i' option tells DFASTAR to treat all lower-case letter and upper-case letters as the same. So that input words "while"and "WHILE" will cause the lexer to return the same terminal symbol number to the parser.

'k' option: Keyword processing.
The 'k' option sets DFASTAR to make use of the keywords specified in the lexical grammar. Keywords are those symbols that start with a slanted quote (`) and end with (`). These are generated by LRSTAR usually. These are the keywords of your language and they are always put into the lexical grammar file (?.lgr). The default operation of DFASTAR is to process these keywords so that the generated lexer will do keyword recognition. This creates larger lexers, sometimes a lot larger than the other possibility of ignoring the keywords. Option 'k=0' will tell DFASTAR to ignore the keywords. This will make smaller lexers. However, then you must use the symbol-table lookup function to recognize the keywords. In the grammar file (?.grm) you must have a statement something like this:

```
<identifier> => lookup()
```

or else your keywords will not be recognized as keywords and will be classified as <identifier>s instead, probably causing a syntax error during testing of the parser.

'ki' option: Keyword and identifier processing.

The 'ki' option sets DFASTAR that you want the generated lexer to recognize <identifier>s and keywords at the same time in the lexer. This means that DFASTAR must use a special algorithm in which it adds extra reductions to states which are necessary to return the correct terminal symbol number for the keywords and also for <identifier>s that are partial keywords. This special algorithm is very fast for all programming languages and it is the default mode of operating (preset as 'ki=1' internally). This setting will handle all cases without problems.

However, if the number of keywords or strings (e.g. “hello”) that you wish to recognize with the lexer, is very large (i.e. 50,000 strings), and your lexer does not need to process <identifier>s, you might want to specify 'ki=0'. In the case of recognizing 42,000 zipcodes, DFASTAR runs for 45 seconds when generating a lexer (with the default: ki=1). When setting 'ki=0' the run time is only 18 seconds. Caution: if you use this option on a lexical grammar that has both keywords and <identifier>s, you will get a lexer that does not recognize <identifier>s make up of partial keywords (e.g. 'whil', 'whi', and 'wh' will cause a syntax error, if the keyword 'while' is one of the valid keywords of your language).

'line' option: Line counting in the lexer.

The 'line' option tells DFASTAR to turn on the @optn_line skeleton code which is usually placed in the skeleton file to include an extra line of code which counts the number of lines in the input stream.

'm' option: Minimize lexer table size.

The 'm' option tells DFASTAR to minimize the size of the lexer tables by doing extra compression during the computation of the displacement array.

'q' option: Quiet mode.

The 'q' option tells DFASTAR to show less information on the screen while running.

's' option: State machine listing.

The 's' option tells DFASTAR to list the original unoptimized state machine in a file. This state machine is the one that you might want to look at when trying to figure out what a conflict error means.

'so' option: State machine optimized listing.

The 'so' option tells DFASTAR to list the optimized state machine in a file.

'tab' option: Tab setting in lexical grammar.

The 'tab' option tells DFASTAR what the tab setting is for the input lexical grammar. This is necessary because indentation is important and the wrong indentation can cause syntax error in the lexical grammar. Also if you send a grammar to someone else who uses a different tab setting, he will not have to spend hours reformatting the lexical grammar to get it to work correctly. This is why DFASTAR forces you to put a tab setting in your lexical grammar. A small inconvenience with a big benefit.

'ts' option: Table-driven small lexer.

The 'ts' option tells DFASTAR to generate the small lexer tables. This is the default table format.

'tm' option: Table-driven medium lexer.

The 'tm' option tells DFASTAR to generate the medium lexer tables. These are usually 10% faster than the small lexer tables.

'v' option: Verbose mode.

The 'v' option tells DFASTAR to display more information on the screen that may be useful to you. The default setting is 'v=0'.

'w' option: Warning messages display.

The 'w' option tells DFASTAR to display the warning message on the screen or in the Output box. The warnings are always displayed in the "?.lgr.log.txt" file for your convenience. However, if you set option 'w', then clicking on a warning message may take you to the file and line number which caused the warning message.

'wait' option: wait for key press.

The 'wait' option tells DFASTAR to wait for someone to press a key after the program has finished. The default setting is 'wait=0' (do not wait for a key press).

More about options 'k' and 'ki'.

Options 'k' and 'ki' are turned on (set to 1) as the default.

The 'k' option tells DFASTAR to use the `keyword`s. These are keywords that start and end with a slanted quote mark (`). These keywords are optional, because you may not want the lexer to do keyword recognition. To avoid keyword processing by the lexer you would specify: 'k=0'.

The default settings (k=1 and ki=1) means that the lexer will recognize all keywords and identifiers in the way you expect for a programming language. This is the easiest way to handle keywords and identifiers. It is also the fastest way to handle them.

However, 'k=1' does not produce the smallest lexers. Languages with many keywords (> 200) will have large lexers when using 'k=1'. The DB2 language (with 550 keywords), would have lexers that are 122K (with 'k=1').

And 'k=1' is not the proper way to handle keywords if you language allows keywords to be used as identifiers (e.g the PL/I language). If keywords are not reserved (like in PL/I), you will want the keywords to be stored in the symbol table (so they can be pointed to by the nodes in the AST). In this case, you should use the option 'k=0'.

With 'k=1', a keyword is recognized as an <identifier> in the lexer and later found to be a keyword by the symbol-table "lookup" function (i.e. <identifier> => lookup() placed in your grammar file). This way the symbol-table "lookup" function will change the type of token from <identifier> to a keyword if it matches a keyword of your language.

The 'ki' option default setting (to 1) is the best solution for most situations. So don't use 'ki=0', unless you understand what problems could occur. The only time you would want to use 'ki=0' is when you are dealing with a huge number of keywords specified by either "..." or '...' or `...`. Then you might want to specify 'ki=0' so that DFASTAR processes your lexical grammar much quicker. You can do this for lexical grammars that have keywords, BUT NO IDENTIFIER defined, such as a list of words, zipcodes or some other of list.

Using 'ki=0' with normal programming languages such as C of Java, will cause a problem with the recognition of identifiers which are partial keywords (e.g. 'whil', 'fo', 'unsign'), these will show up in the parser as <error> symbols and cause a syntax error.

There is a built-in check for this and the other parser-lexer synchronization problems in the parser skeleton code. So you don't have to worry about it, you will get some error messages telling you about the problem.

Chapter 4.
Output Files.

DFASTAR always creates five output files when analyzing a grammar. For example, if your grammar is **calc.grm**, then you will get these five output files:

calc.lgr.conflicts.txt	A conflicts file, showing the conflicts in your grammar.
calc.lgr.grammar.txt	A grammar file, showing your grammar and the rule numbers.
calc.lgr.log.txt	A log file, showing a log of what happened.
calc.lgr.states.txt	A state-machine file, showing the lexer state machine.

Chapter 5.
Grammar Introduction

Definition

LBNF means Lexical BNF. It is a BNF, Backus Normal Form, that completely defines the tokens of your language. Note that LRSTAR will create the tokens for this file and assign terminal symbol numbers, corresponding to the terminal numbers defined in the grammar file (?.grm). If the lexical grammar (?.lgr) already exists, then LRSTAR will rewrite the token list and assign the current terminal symbol numbers.

Modern Concepts In LBNF

1. Angled-bracket symbols to express variable symbols, such as <identifier>.
2. Extended BNF grammar notation or regular expressions in your grammar rules.
3. Escape symbols that you can define to mean whatever you want (e.g. \z, \n, \t …).
4. Well known string notation using single or double quotes (e.g. “hello”, 'there').
5. Special notation for the keywords of your language indicated by slanted quote marks (e.g. `while`).

Special keyword notation

This special keyword notation using slanted quote marks (e.g. `if`, `then`, `else`) is useful when you also have <identifier> defined in the lexical grammar. LRSTAR automatically generates all keywords of your language into the lexical-grammar file along with the appropriate terminal symbol number to make your lexer compatible with the parser.

The advantage of this concept is that you now have two choices: (1) Let the lexer do keyword recognition (option 'ki', the default) or (2) Let the symbol-table lookup() function do keyword recognition (option 'k=0'). Note, if you want the symbol-table to do keyword lookup, then you will have to place this kind of statement in the grammar (?.grm) file:

<identifier> => lookup(); // Do symbol-table lookup for <identifier>s.

Having the keywords recognized by the lexer is the most reliable way to proceed (this is the default). However, if you want smaller lexers, you may have the symbol-table lookup function do the keyword recognition. Also note that certain languages allow the keywords to be used as variable names (<identifier>s) (e.g. PL/I allows this). In these weird languages you will probably want to put the keywords into the symbol table (option 'k=0').

Chapter 6.
Lexical-Grammar Symbols

Lexical-Grammar Symbols

Lexical-grammar symbols are the “words” of your grammar which define the tokens of your language. Here is a list of the different types of symbols allowed in a lexical grammar:

<alpha>
<alpha> -> (letter | '_') (letter | digit | '_')*

Used as nonterminals, terminals, keywords, action names, node names, defined constants, arguments.
Examples: GotoStmt, Input_file, _int64_.

<lexical>
<lexicon> -> '<' <alpha> '>'

Used as terminals, arguments.
Examples: <identifier>, <string>, <eof>, <end_of_file>, <_int64_>

<literal>
<literal> -> '" LChar+ '"
LChar -> 33..255 - \n - \z

Used as terminals, arguments.
Examples: ‘hello’, ‘>’, ‘>=’, ‘TRUE’.

<string>
<string> -> '"' SChar+ '"'
SChar -> 33..255 - '"' - \n - \z

Used as arguments only.
Examples: “helloworld”, “if”, “integer”, “float”.

<keyword>
<keyword> -> '`' KChar+ '`'
KChar -> 'a'..'z' | 'A'..'Z'

Used as arguments only.
Examples: `helloworld`, `if`, `integer`, `float`.

<integer>
<integer> -> digit+

Used as arguments.
Examples: 0, 1, 9, 100, 255, 65535.

Grammar Operators and Punctuators

Grammar operator and punctuators are used to further define the syntax of your computer language. Here is a list of the different types of operators and punctuators allowed in a grammar:

Rule Notation

'=>' means “outputs” or “returns” a token number or defined constant.
'->' means “is a“ (for a rule which defines the input for this token).
'[]' means “bypass“ this token (nothing is returned to the calling function).

'|' means “alternate choice” for another character.
'..' means “up to and including” (a range of characters).
'-' means “subtract” the next character or range from the set being defined.

EBNF Operators

'...' means “one or more occurrences”.

'(' means “group start”.
')' means “group end”.

'[' means “optional start”.
']' means “optional end”.

'|' means “or”, inside of a group (e.g. (a | b | c)).

Regular Expression Operators

'+' means “one or more occurrences”.
'*' means “zero or more occurrences”.
'?' means “optional occurrence”.

Comments

Comments are allowed in the same style as C++. Line comments start with '//' and end with the end-of-line character, for example:

```
// C++ Grammar.
// by John Smith.
// in August 2007.
```

Block comments start with '/*' and end with '*/'. These cannot be nested (one inside of another). Here is an example:

```
/* Removed 10-15-04 NKP
CreateStmt -> CREATE TableName ColStuffList ';'
           -> CREATE TableName UniqueStuff ';' // Added 09-15-04 JKL
*/
```