Skeleton Notation

for

LRSTAR 4.0

DFASTAR 4.0

April 2013

Chapter 1
Skeleton Notation

Skeleton.

The parser skeleton, (e.g. **parser.cpp.skl**), contains the parser code and skeleton codes. To make the parser skeleton suitable for rewriting in other programming languages, unique codes were designed. These codes are placed in the parser source code at the appropriate places. They tell LRSTAR how to create the parser, which code to include and which to discard, thereby giving you a custom parser depending on the options you supplied on the command line. .

The following section explains the format of these skeleton codes and gives examples of how to use them. Hopefully, you will not have to learn any of this tedious stuff and just use the 'parser.cpp.skl' without learning the skeleton code names. If, however, you want to rewrite the skeleton in another programming language, you may have to learn some of these details.

Command Line.

The skeleton file (e.g. parser.cpp.skl) is put in the command line as follows:

```
lrstar c.grm parser.cpp.skl parser.cpp
```

Skeleton Code Short Forms.

The short skeleton codes have several possible formats, as follows:

```
@//                                     (Single line comment)

@/*                                     (Multiple-line comment start)
@*/                                     (Multiple-line comment end)

@ name ;                                (Single usage, string format)
@ name operator ;                       (Single usage, operator indicates format)
@ name conditional_operator ;           (Single line condition)
@ name conditional_operator ; ...       (Multiple-line condition start)

expression ; ...                        (Multiple-line condition start)
expression .and. expression ; ...       (Multiple-line condition start)
expression .or.  expression ; ...       (Multiple-line condition start)
expression .eq.  expression ; ...       (Multiple-line condition start)
expression .ne.  expression ; ...       (Multiple-line condition start)
expression .gt.  expression ; ...       (Multiple-line condition start)
expression .ge.  expression ; ...       (Multiple-line condition start)
expression .lt.  expression ; ...       (Multiple-line condition start)
expression .le.  expression ; ...       (Multiple-line condition start)
@@                                      (Multiple-line condition end)
```

@//

The '@.' means line-comment start. When LRSTAR encounters this in the skeleton it ignores the rest of the line. This means that all the text between the '@.' and the end-of-line will not be transferred to the generated parser.

@/*

The '@/*' means multi-line-comment start. When LRSTAR encounters this it ignores the rest of the line and all following lines until it finds a comment terminator '@*/. All the lines in between the '@/*' and the '@*/' will not be transferred to the generated parser.

@*/

The '@*/' means terminate this comment. When LRSTAR encounters this it stops ignoring text. The rest of the line that contains the '@*/' will be ignored and regular processing will start with the beginning of the next line. Some examples of comments follow:

```
@//  Author:  John Smith, 1995.
@//  Included for error recovery.
@/*
    This skeleton contains:
    Error recovery code.
    Abstract-Syntax-Tree code.
    Help message code.
@*/
```

name

The name must be one of the legal skeleton variable names mentioned later. It can be used to select a string, such as the parser filename, or a number, such as the size of the accessor array, or a list of numbers, such as the production lengths. For example:

```
@prod_leng
```

The default for just a name is string type (.s). The following would be just the grammar name:

```
@grm_name;
```

operator

A name can be followed by an operator. An operator can be one of the following:

.d	Decimal form: the number of elements in the list.
.s	String or character form: the text associated with the @code.
.t	Type form: the type of the @code.
.<number>d	Decimal form with size of space allotted for the number.
.<number>s	String form with size of space allotted for the string.
.<number>t	Type form with the size of the space allotted for the type.

Examples:

```
@term_symb.d;            (number of terminal symbols)
@grm_name.20s;           (grammar name within a 20 character space)
@pact_arg.  8t;          (parse action argument type)
```

conditional_operator

The name may be followed by a conditional_operator. There are two conditional operators as follows:

```
?          (if exists, more than zero elements)
!          (if does not exist, zero elements)
```

Examples

```
@pact_numb?;
@pact_numb!;

@pact_numb?;...
   if (pact_numb [p] >= 0)
   {
      /* some code */
   }
@pact_arg?;...
   if (pact_arg [p] >= 0)
   {
      /* some code */
   }
@@
@@
```

You may have a conditional group nested inside of another conditional group, as shown above.

expression

An expression has the following formats:

```
@ name ?                         (Same as .ge.1)
@ name !                         (Same as .eq.0)
@ name <operator> <number>
```

Expressions can be joined together with the following <operator>s:

.and.	"and"
.or.	"or"
.eq.	"equal to"
.ne.	"not equal to"
.gt.	"greater than"
.ge.	"greater than or equal to"
.lt.	"less than"
.le.	"less than or equal to"

Here are some examples:

```
@optn_debug.ge.1;...
/* some code ... */
@@

@optn_debug.ge.1.and.@make_ast?;...
/* some code ... */
@@
```

```
@nact_arg.ge.1.or.@pact_arg.ge.1;...
/* some code ... */
@@
```

Skeleton Code Long Form.

The long form of skeleton code is used to indicate an array of numbers or a list of names or an array of function pointers. The long form always indicates a list of elements rather than just one value. The long form is different from the short form as follows:

```
@ name . count | format | sep | endofline |;
```

Now let's build a complete skeleton code one piece at a time and see how the pieces fit together.

name

As defined previously for the short form of skeleton code. An @-sign followed by a name. Let's use @tail_numb for an example.

```
@tail_numb
```

count

Count specifies the number of items per line:

```
@tail_numb.10
```

format

Format specifies the format of the number or string:

```
@tail_numb.10|%5d|
```

sep

Sep is a separator to be placed between each item on a line. If a '.10' expression is used, then there will be nine separators used to separate the symbols. After the 10th item the end-of-line separator is used (see below). For separator we use a comma and a space, so now we have:

```
@tail_numb.10|%5d|, |
```

end-of-line

End-of-line is used to indicate the end-of-line separator on all but the last line. To indicate a separator for the last line just put it after the last |; If we use a comma and a newline character, we now have:

```
@tail_numb.10|%5d|, |,\n|;
```

This is a complete long form skeleton code. It could be simplified or expanded depending on the needs of your program. Here is another example:

```
@tail_numb.10|%5d|, |,\n      |;
```

Here is a form used for string data, like symbol names:

```
@term_symb.1|"%s"|, |,\n      |;
```

Here is a form that could be used for case statements:

```
@oact_name.1|case %3d: %s(); break;||\n |;
```

Special Codes

There are additional operators allowed (e.g. %g) in the long form of the skeleton code, as shown below:

```
@nact_func.1|%g._ASTAction::%s|,|,\n         |;
```

The %g is replaced by the grammar name. Here is a list of special codes that are usable in the long form of the skeleton code:

```
%d                (a number associated with the skeleton name)
%g                (the grammar name)
%s                (a string associated with the skeleton name)
```

Here is an example skeleton code usage:

```
@def_cons?;...
enum tokens
{
   @def_cons.1|%s = %d||,\n      |;
};
@@
```

Here is the code that was generated:

```
enum tokens
{
   IDENT = 1,
   INTEGER = 2,
   EOFILE = 3,
   PROGRAM = 4,
   IF = 5,
   THEN = 6,
   ELSE = 7,
   ENDIF = 8
};
```

Function Calls Instead of Function Pointers

Here is another example which may be useful for Java programmers, who want to re-write the parser skeleton in Java. You may want to use actual function calls instead of function pointers. This is what the skeleton code should look like (note the backslash (\) placed before the ';'):

```
switch (n)
{
   @nact_func.1|case %d: %s (n)\; break\;||\n      |;
}
```

Here is what LRSTAR will generate from the above code:

```
switch (n)
{
   case 0: goal_ (n); break;
   case 1: program_ (n); break;
   case 2: store_ (n); break;
   case 3: if_ (n); break;
   case 4: target_ (n); break;
   case 5: eq_ (n); break;
   case 6: ne_ (n); break;
   case 7: add_ (n); break;
   case 8: sub_ (n); break;
   case 9: mul_ (n); break;
   case 10: div_ (n); break;
   case 11: int_ (n); break;
   case 12: ident_ (n); break;
   case 13: then_ (n); break;
   case 14: else_ (n); break;
   case 15: then2_ (n); break;
   case 16: else2_ (n); break;
}
```

LRSTAR Skeleton Keywords

```
"accp_sta",    "dt",  ACCP_STA,      "Accept (final) state number",
"arg_numb",    "dt",  ARG_NUMB,      "Argument numbers",
"arg_text",    "dt",  ARG_TEXT,      "Argument text strings",
"bmat_col",    "dt",  BMAT_COL,      "B matrix column index",
"bmat_mask",   "dt",  BMAT_MASK,     "B matrix mask",
"bmat_numb",   "dt",  BMAT_NUMB,     "B matrix numbers",
"bmat_row",    "dt",  BMAT_ROW,      "B matrix row index",
"def_cons",    "dt",  DEF_CONS,      "Defined constants",
"eof_numb",    "dt",  EOF_NUMB,      "End of file terminal number",
"eol_numb",    "dt",  EOL_NUMB,      "End of line terminal number",
"err_used",    "dt",  ERR_USED,      "<error> symbol used in grammar (0 or 1)",
"grm_file",    "s ",  GRM_FILE,      "Grammar filename",
"grm_name",    "s ",  GRM_NAME,      "Grammar name",
"grm_text",    "s ",  GRM_TEXT,      "Grammar text (whole file contents)",
"head_prod",   "dt",  HEAD_PROD,     "Head symbol first production (rule)",
"head_symb",   "dt",  HEAD_SYMB,     "Head symbol for a production (rule)",
"key_numb",    "dt",  KEY_NUMB,      "<keyword> symbol number",
"make_ast",    "dt",  MAKE_AST,      "Make AST switch (0 or 1)",
"nact_arg",    "dt",  NACT_ARG,      "Node action index into argument numbers list",
"nact_func",   "dt",  NACT_FUNC,     "Node fuction names",
"nact_numb",   "dt",  NACT_NUMB,     "Node action number for a production (rule)",
"nd_action",   "dt",  ND_ACTION,     "Nondeterministic action list (goto or reduce)",
"nd_start",    "dt",  ND_START,      "Nondeterministic start point in action list",
"nd_term",     "dt",  ND_TERM,       "Nondeterministic terminal symbol list",
"nmat_col",    "dt",  NMAT_COL,      "N matrix column index",
"nmat_numb",   "dt",  NMAT_NUMB,     "N matrix numbers",
"nmat_row",    "dt",  NMAT_ROW,      "N matrix row index",
"node_arg",    "dt",  NODE_ARG,      "Node index into argument numbers list",
"node_name",   "dt",  NODE_NAME,     "Node names",
"node_numb",   "dt",  NODE_NUMB,     "Node number for a production",
"numb_head",   "dt",  NUMB_HEAD,     "Number of head symbols (nonterminals)",
"numb_nact",   "dt",  NUMB_NACT,     "Number of node actions",
"numb_node",   "dt",  NUMB_NODE,     "Number of nodes",
"numb_pact",   "dt",  NUMB_PACT,     "Number of parse actions",
"numb_prod",   "dt",  NUMB_PROD,     "Number of productions (rules)",
"numb_sta",    "dt",  NUMB_STA,      "Number of states",
"numb_tact",   "dt",  NUMB_TACT,     "Number of token actions",
"numb_term",   "dt",  NUMB_TERM,     "Number of terminal symbols",
"optn_debug",  "dt",  OPTN_DEBUG,    "Debug parser option (0, 1, 2, 3)",
"optn_exp",    "dt",  OPTN_EXP,      "Expecting list option (0 or 1)",
"optn_fc",     "dt",  OPTN_FC,       "Function calls instead of function pointers",
"optn_na",     "dt",  OPTN_NA,       "Number of node arguments allowed for a node",
"optn_nd",     "dt",  OPTN_ND,       "Nondeterministic option (0, 1, 2)",
"out_file",    "s ",  OUT_FILE,      "Output filename (e.g. parser.cpp)",
"pact_arg",    "dt",  PACT_ARG,      "Parsing action index into argument numbers list",
"pact_func",   "dt",  PACT_FUNC,     "Parsing action function names",
"pact_numb",   "dt",  PACT_NUMB,     "Parsing action numbers (one for each production)",
"prod_head",   "dt",  PROD_HEAD,     "Production head symbol (one for each rule)",
"prod_leng",   "dt",  PROD_LENG,     "Production length",
"prod_revs",   "dt",  PROD_REVS,     "Production reverse code (0 or 1)",
"prod_tail",   "dt",  PROD_TAIL,     "Production tail (first tail number)",
"program",     "s ",  PROGRAM,       "Program name (e.g. LRSTAR)",
"restore",     "dt",  RESTORE,       "Restore parser stack code (0 or 1)",
"rmat_col",    "dt",  RMAT_COL,      "R matrix column index",
"rmat_numb",   "dt",  RMAT_NUMB,     "R matrix numbers",
"rmat_row",    "dt",  RMAT_ROW,      "R matrix row index",
"skl_file",    "s ",  SKL_FILE,      "Skeleton filename (e.g. parser.cpp.skl)",
"sta_accs",    "dt",  STA_ACCS,      "State accessor symbols",
"sta_dot",     "dt",  STA_DOT,       "State dot (in production)",
"sta_item",    "dt",  STA_ITEM,      "State items",
"sta_prod",    "dt",  STA_PROD,      "State productions",
"tact_arg",    "dt",  TACT_ARG,      "Terminal action index into argument numbers list",
"tact_func",   "dt",  TACT_FUNC,     "Terminal action function names",
"tact_numb",   "dt",  TACT_NUMB,     "Terminal action numbers (one for each terminal)",
"tail_numb",   "dt",  TAIL_NUMB,     "Tail numbers",
"term_symb",   "dt",  TERM_SYMB,     "Terminal symbols (all terminals)",
"tmat_col",    "dt",  TMAT_COL,      "T matrix column index",
"tmat_numb",   "dt",  TMAT_NUMB,     "T matrix numbers",
"tmat_row",    "dt",  TMAT_ROW,      "T matrix row index",
"version",     "s ",  VERSION,       "Version of program (e.g. 3.0.100)",
```

DFASTAR Skeleton Keywords

```
"bmat_col",   "dt",   BMAT_COL,    "B matrix column index",
"bmat_numb",  "dt",   BMAT_NUMB,   "B matrix (0's and 1's)",
"bmat_row",   "dt",   BMAT_ROW,    "B matrix row index",
"def_cons",   "dt",   DEF_CONS,    "Defined constants",
"err_token",  "dt",   ERR_TOKEN,   "Error token",
"grm_file",   "s ",   GRM_FILE,    "Grammar filename",
"grm_name",   "s ",   GRM_NAME,    "Grammar name",
"grm_text",   "s ",   GRM_TEXT,    "Grammar text (complete file contents)",
"numb_sta",   "dt",   NUMB_STA,    "Number of states",
"numb_term",  "dt",   NUMB_TERM,   "Number of terminals",
"optn_code",  "dt",   OPTN_CODE,   "Code ('c') option specified",
"optn_col",   "dt",   OPTN_COL,    "Column number in lexer",
"optn_debug", "dt",   OPTN_DEBUG,  "Debug option",
"optn_line",  "dt",   OPTN_LINE,   "Line number in lexer",
"optn_medium","dt",   OPTN_MEDIUM, "Medium table-driven lexer",
"optn_small", "dt",   OPTN_SMALL,  "Small table-driven lexer",
"out_file",   "s ",   OUT_FILE,    "Output filename",
"program",    "s ",   PROGRAM,     "Program name (DFASTAR)",
"skl_file",   "s ",   SKL_FILE,    "Skeleton filename",
"strings",    "dt",   STRINGS,     "Strings for return values",
"term_numb",  "dt",   TERM_NUMB,   "Terminal number for lexer tables",
"tmat_col",   "dt",   TMAT_COL,    "T matrix column index",
"tmat_numb",  "dt",   TMAT_NUMB,   "T matrix numbers",
"tmat_row",   "dt",   TMAT_ROW,    "T matrix row index",
"version",    "s ",   VERSION,     "Version of program",
```

Skeleton name examples.

Some examples are shown here.

@term_symb

The list of terminal symbols. These are a necessary part of the skeleton because all these symbols have to be hashed into the symbol table before parsing begins. For example:

```
char *T_symbol [@term_symb.d;] =
{
   @term_symb.1|"%s"|,|,\n      |;
};
```

@term_lex

A list of text strings giving the lexicon symbols of the grammar with the '<' and '>' characters. For example:

```
@term_lex.1|   %20s |||;
```

@term_oper

A list of text strings giving the operators and punctuation symbols of the grammar indicated by being enclosed within single quotes. Those symbols enclosed within double quotes are not available as a separate list. For example:

```
@term_oper.1|   %20s |||;
```

@term_key

A list of text strings giving the keywords of the grammar grammar. This will include all alpha symbols that are not head (nonterminal) symbols and will include also those symbols enclosed within double quotes. For example:

```
@term_key.1|   %20s |||;
```

@head_symb

The list of head (nonterminal) symbols of the grammar. For example:

```
char *N_symbol [@head_symb.d;] =
{
   @head_symb.1|"%s"|,|,\n      |;
};
```

@head_prod

A list of the first production number for a head (nonterminal) symbol. For example:

```
char *N_symbol [@head_prod.d;] =
{
   @head_prod.1|%6d|,|,\n      |;
};
```

@prod_leng

The production (rule) lengths (zero based: 0, 1, 2, ...). For example:

```
signed char PL [@prod_leng.d;] =
{
   @prod_leng.10|%6d|,|,\n      |;
};
```

@prod_head

The Head symbol number for each production. Each production (rule) begins with a head (nonterminal) symbol and some productions within a group defining a particular head symbol have the same head as other productions in that group. This data is not needed by the parser, but it may be useful for special things like debugging. Sample usage typically found in a parser skeleton:

```
int HS [@prod_head.d;] =
{
   @prod_head.10|%6d|,|,\n      |;
};
```

@prod_tail

The index of the first tail symbol for a productions. This is not necessary for the parser, but it might be useful for something else, such as error recovery. You may find them useful if you want to display the production being reduced. You may also want to use them to display the whole grammar for the user. For example:

```
int f_tail [@prod_tail.d;] =
{
   @prod_tail.10|%6d|,|,\n      |;
};
```

@prod_revs

The code (0 or 1) which indicates whether or not a production's node numbers currently on the parse stack should be reversed when added to the AST. For example:

```
int reverse [@prod_revs.d;] =
{
   @prod_revs.10|%6d|,|,\n      |;
};
```

@tail_numb

The tail-symbol numbers for all the productions. Note, terminal-symbol numbers are zero or positive and nonterminal-symbol numbers are negative. The goal symbol of the grammar is zero, but cannot be a tail symbol of the grammar. For example:

```
int tail [@tail_numb.d;] =
{
   @tail_numb.10|%6d|,|,\n      |;
};
```

@tact_numb

The input token action number corresponding to the action name for each terminal symbol. If a number is -1, there is no action for that terminal symbol. For example:

```
int ta_numb [@tact_numb.d;] =
{
   @tact_numb.10|%6d|,|,\n      |;
};
```

@tact_arg

The token action argument index into the complete list of arguments. For example:

```
int ta_arg [@tact_argi.d;] =
{
   @tact_argi.10|%6d|,|,\n      |;
};
```

@pact_func

The parse action function names. For example:

```
extern int (*pact_func [@pact_func.d;]) () =
{
   @pact_func.1|%s|,|,\n      |;
};
```

@pact_numb

The parse action numbers corresponding to the parse action names. One for each production. If -1, there is no action for the production. For example:

```
int pa_numb [@pact_numb.d;] =
{
   @pact_numb.10|%6d|,|,\n      |;
};
```

@pact_arg

The index of the first argument for a parse action. There are n of these, where n is the number of productions. You cannot get the number of arguments. The function will have to know how many arguments there are (a fixed number):

```
int f_arg [@pact_arg.d;] =
{
   @pact_arg.10|%6d|,|,\n      |;
};
```

@arg_numb

The output arguments as listed after the action-name/node-name in the grammar. These are string data (%s) or numeric data (%d). This is because some of them may be defined constants.

```
int arg [@arg_numb.d;] =
{
    @arg_numb.10|%s|,|,\n      |;
};
```

@arg_type

The output argument types. For example:

```
int arg_type [@arg_type.d;] =
{
    @arg_type.10|%6d|,|,\n      |;
};
```

@arg_text

The text for those arguments that are strings. For example:

```
int text [@arg_text.d;] =
{
    @arg_text.1|"%s"|,|,\n      |;
};
```

@def_cons

The defined constants used in the grammar as arguments for output actions or node names. Code as follows:

```
@arg_defs?;...
      @arg_defs.1|#define %20s %3d||\n|;
@@
```

@node_name

The node names, defined by the operator '+>'. Two examples follow:

```
@node_name.1|#define %20s %3d||\n|;

int node_name [@node_name.d;] =
{
   @node_name.1|"%s"|,|,\n      |;
};
```

@nact_func

The node action function names.

```
// Node action function pointers.
short (*@grm_name;_parser::nact_func [@nact_func.d;]) (int n) =
{
   @nact_func.1|%g._ASTAction::%s|,|,\n         |;
};
```

@bmat_numb

The bit matrix or test matrix that determines if the current input terminal symbol should cause a shift action or not. This matrix is indexed by the state number and the terminal symbol number. For example:

```
int Bm [@bmat_numb.d;] =
{
    @bmat_numb.10|%6d|,|,\n      |;
};
```

@bmat_row

The row (or displacement for a row) of numbers corresponding to a state in the B-matrix. For example:

```
int Br [@bmat_row.d;] =
{
    @bmat_row.10|%6d|,|,\n      |;
};
```

@bmat_col

The column in the current row in the B-matrix which gives the actual byte or bit. For example:

```
int Bc [@bmat_col.d;] =
{
    @bmat_col.10|%6d|,|,\n      |;
};
```

@tmat_numb

A Terminal transition matrix that gives the next state number. For example:

```
int Tm [@tmat_numb.d;] =
{
   @tmat_numb.10|%6d|,|,\n      |;
};
```

@tmat_row

An array of numbers indexed by the current state number, which gives the row displacement in the T-matrix. For example:

```
int Tr [@tmat_row.d;] =
{
    @tmat_row.10|%6d|,|,\n      |;
};
```

@tmat_col

An array of numbers indexed by the current terminal symbol, which gives the exact integer to examine in the T-matrix. For example:

```
int Tc [@tmat_col.d;] =
{
    @tmat_col.10|%6d|,|,\n      |;
};
```

@nmat_numb

A Nonterminal transition matrix that gives the next state number. For example:

```
int Nm [@nmat_numb.d;] =
{
    @nmat_numb.10|%6d|,|,\n      |;
};
```

@nmat_row

An array of numbers indexed by the state returned to when doing a reduction, which gives the row displacement in the N-matrix. For example:

```
int Nr [@nmat_row.d;] =
{
    @nmat_row.10|%6d|,|,\n      |;
};
```

@nmat_col

An array of numbers indexed by the production number being reduced, which gives the exact integer to examine in the N-matrix. For example:

```
int Nc [@nmat_col.d;] =
{
    @nmat_col.10|%6d|,|,\n      |;
};
```

@rmat_numb

A Reduction matrix that gives the reduction to make if the a default reduction was not indicated. This is only accessed for multiple reduce states and the final state (of goal reduction state). For example:

```
int Rm [@rmat_numb.d;] =
{
    @rmat_numb.10|%6d|,|,\n      |;
};
```

@rmat_row

An array of numbers indexed by the current state number, which gives the row displacement for the R-matrix. For example:

```
int Rr [@rmat_row.d;] =
{
    @rmat_row.10|%6d|,|,\n      |;
};
```

@rmat_col

An array of numbers indexed by the current terminal symbol, which gives the exact integer to examine in the R-matrix. For example:

```
int Rc [@nmat_col.d;] =
{
   @nmat_col.10|%6d|,|,\n      |;
};
```

@numb_sta

The number of states in the state machine, numbered 0 through n. For example:

```
static int n_states @numb_sta.6d;;
```

@numb_term

The number of terminal symbols. For example:

```
static int n_terms @numb_term.6d;;
```

@grm_file

Is the filename of the input grammar, for example:

```
COBOL.GRM
```

@grm_name

Is the name of the input grammar, for example:

```
COBOL
```

@skl_file

Is the filename of the input skeleton file.

```
parser.cpp.skl
```

@out_file

Is the filename of the output parser tables. For example, we might have:

```
Generated by:             @program; @version;
Input grammar:            @grm_file;
Input skeleton:           @skl_file;
Output parser:            @out_file;
```

Which might generate something like this:

```
Generated by:             LRSTAR 2.2.005
Input grammar:            cobol.grm
Input skeleton:           parser.cpp.skl
Output tables:            parser.cpp
```