LRSTAR 4.0

User's Manual

April 2013

Acknowledgments

Since many of the ideas used in the development of LRSTAR 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 LRSTAR.

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 LRSTAR software is supplied in a compressed file format, such as a 'zip' file. For example the download file may be called, **LRSTAR.4.0.000.zip**. After you download this file, double-click on it to run your unzip program. Then click “Extract” to extract the contents to a directory (folder) of your choice, usually C:\.

The “LRSTAR.4.0.000” folder.

The unzipped folder name in this case should be: **LRSTAR.4.0.000**. Inside this main folder, you should have 1 or more folders corresponding to how many programs you downloaded and unzipped:

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)

The doc folder.

Inside the 'doc' folder you should have:

definition_of_terms.pdf
lrstar_users_manual.pdf
skeleton_notation.txt
tbnf_paper.pdf
dfastar.readme.txt
lrstar.readme.txt
software_license.txt

The grm folder.

Inside the 'grm' folder you should have:

*.grm files (20 or more grammar files)

The project folder.

Inside the 'project' folder you should have 6 sample projects as follows:

c
calc
calc1
ielr
lr1
typedef

The skl folder.

Inside the 'skl' folder you should have:

*.skl files	(the parser and lexer skeleton files)

Step 1. Pick a project.

To get started, in the '**parser**' folder, pick a project such as **'calc'** and inside this folder you should have a file:

'workspace.vcproj.PAULBMANN.Owner.user'	(Visual Studio 2008 file)

Rename this file to have your computer name and your user name such as:

'workspace.vcproj.MYCOMPUTER.MyUserName.user'	(Visual Studio 2008 file)

This is necessary (for Visual Studio 2008) to retain the preset settings for the build and test. If you have Visual Studio 2010, you may not have to do this, because it will rename this file for you.

Step 2. Start Visual Studio.

You should also have this file in the same directory (folder):

'workspace.vcproj'	(Visual Studio 2008 file)

Double-click on **'workspace.vcproj'** and your Visual Studio program will start and populate the solutions workspace with a list of relevant files for this **'calc_parser'** project.

Step 3. Add File Extensions "grm", "skl", "lgr" to Visual Studio C++

Go into the Tools/Options/Text Editor/File extensions/ and include the "grm", "skl", "lgr" file extensions as Microsoft Visual C++ file types. Now you will have syntax coloring for all skeleton files. But the best thing is that you will be able to double-click on a parse-action or node-action in your grammar and be taken to the C++ function associated with it. For example:

```
Stmt ~> Target '=' Exp ';'    *> store_
```

If you double click on the '**store_**' you will go to the function '**store_**' in your source code. Very useful!

Step 4. Set Read-Only Files NOT to Allow Editing.

Go to Tools/Options/Environment/Documents/ and unselect the 3rd little box from the top, which says :

[   ] Allow editing of read-only file; warn when attempt to save.

Make sure this is unchecked as shown above. Then you will not waste any time making changes to a file that was generated automatically. All generated files are read-only.

Step 5. Build The Calc Parser.

Under the **'Build'** menu, do a **'Rebuild Solution'** to make sure everything is working correctly. A working '**calc_parse.exe**' program should have been created. Next, under the **'Debug'** menu, do a **'Start Without Debugging'** and the parse program should run and read the '**text.input.txt**' file. Now, look at the '**text.output.txt**' file which was created and see if it looks like everything worked correctly. You should study all the files to learn how the system works. If things are not working correctly, contact technical support via the website, Compilerware.com.

Chapter 2.
Command line syntax.

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

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

Where 'lrstar' is the program name (the 'lrstar.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.grm' 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: 'parser.cpp.skl'.

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

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

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

Here are some examples:

```
lrstar ansic.grm
lrstar ansic parser.cpp.skl parser.cpp /g /k /w /s
lrstar ansic parser.cpp.skl parser.cpp parser.h.skl parser.h /v /w /eof=0
```

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 '**lrstar**' without any arguments then you will get a display showing the program options, as follows:

```
LRSTAR 4.0.000 Copyright 2013 Compilerware.
|
|   LALR(k) Parser Generator
|
|   lrstar <grammar> [<skeleton> <output>]... [/<option>...]
|
|   OPTION  DESCRIPTION                               DEFAULT
|   a       Analyze the grammar only.                    0
|   ct      Conflict traceback information.              0
|   d       Debug option turned on in parser (1,2,3).    0
|   e=?     Error count before quitting.                10
|   exp     Expecting symbol list for syntax errors.     0
|   eof     End-of-file included in .lgr file.           1
|   fc      Function calls instead of function ptrs.     0
|   g       Grammar listing output.                      0
|   h       HTML grammar listing output.                 0
|   i       Ignore node names and node actions.          0
|   k       Keywords checking and listing.               0
|   lalr    LALR(1) grammar analysis.                    0
|   m       Minimize parser-table size.                  0
|   na      Number of node arguments.                    1
|   nd      NonDeterminism for RR conflicts.             0
|   nd=2    NonDeterminism for RR and SR conflicts.      0
|   o       Optimize (chain-reduction elimination).      0
|   q       Quiet mode, minimal screen display.          0
|   rr      Show reduce-reduce conflicts only.           0
|   s       State machine listing.                       0
|   slr     SLR(1) grammar analysis.                     0
|   tab=?   Tab setting in grammar.                      0
|   v       Verbose mode, show extra messages.           0
|   w       Warning messages display.                    0
|   wait    Wait on key press to end program.            0
|
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 LRSTAR program options. Here is the list of options.

Option	Description	Default Value
a	Analyze the grammar only.	0
ct	show Conflict Traces in conflicts.txt file.	0
d	put level 1 Debug code in parser.	0
e=?	Error count before quitting.	10
eof	put End-Of-File character in **.lgr** file.	1
exp	put Expecting list code in parser.	0
fc	Function calls instead of function pointers.	0
g	Grammar listing output.	0
h	HTML grammar output.	0
i	Ignore AST operations in grammar.	0
k	do Keyword checking in grammar.	0
m	Minimize parser table size.	0
lalr	LALR(1) grammar analysis.	0
na	Number of node arguments for each node name.	0
nd	Nondeterministic parsing for RR conflicts.	0
nd=2	Nondeterministic parsing for RR and SR conflicts.	0
o	Optimize parser table for speed.	0
q	Quiet mode, minimal screen display.	0
rr	show Reduce-Reduce conflicts only.	0
s	show State machine listing.	0
slr	SLR(1) grammar analysis.	0
tab=?	Tab setting in grammar file.	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

Options that affect the parser code:

d	put Debug code in parser.	0
exp	put Expecting list code in parser.	0
i	Ignore AST operations in grammar.	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 LRSTAR 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).

'ct' option: Conflict traces.

This option will ask LRSTAR to show conflict traces. It shows how the different choices of shift and reduce act during parsing. The shift action is the default, unless an override was specified with the '/->' operator at the beginner of the rule which has higher priority. If you specify the option 'slr' the 'ct' option will be ignored, because SLR(1) grammar analysis does not provide the internal structures which are needed to give a conflict trace.

'd' option: Debug information in output.

The 'd' option tells LRSTAR to generate a parser which will display extra information while it is running, information that is useful for debugging purposes. LRSTAR sets the skeleton code, @optn_debug to 1 so that the skeleton parser will contain the source code to create this debugging information. The current 'parser.cpp.skl' file has code that supports d=1, d=2 and d=3. d=1 gives the least amount of information and d=3 the most.

'e=?' option: Error count limit.

The 'e=?' option tells LRSTAR how many errors to show before quitting. However, some errors will stop the LRSTAR program before listing any other errors. The default number of errors to display is 10. To change to 20 errors, use 'e=20'.

'eof' option: EOF character in lexer grammar.

The 'eof' option tells LRSTAR to include the <eof> terminal symbol in the lexical grammar file which it will create (if nonexistent) or modify. When LRSTAR processes a x.grm file, it will create or modify the x.lgr file, so the x.lgr file will have the correct terminal symbol numbers needed by the parser. LRSTAR will include all terminal symbols of the grammar except the semantic ones ({typedef}, {pgmname}, etc.).

The default is to include the <eof> terminal symbol in this list in the x.lgr file. If you do not want the <eof> to be listed, specify this: 'eof=0'. In this case you will have to change the way the lexer terminates upon encountering an end-of-file condition. The fastest lexer speed will be obtained by including the <eof> symbol in the x.lgr file.

'exp' option: Expecting list in case of error.

The 'exp' option tells LRSTAR to include the syntax analysis code in the generated parser. This syntax analysis will generate the list of tokens expected at the error point. This is very useful to someone using your computer language. This will cause a slight decrease in parsing speed, maybe about 1% decrease. The default is 'exp=0' (do not include the expecting list cod in the parser). Here is a sample expecting list from the calc parser. The 'exp=2' generates a parser than also displays the contents of the parse stack as well as the following.

```
test.inp(6) : Error          if == 0
test.inp(6) : Error ------------^ at == '=='

Expecting one of the following:

      <identifier>
      <integer>
      '('
```

'fc' option: Function calls.

The 'fc' option sets the skeleton code **@optn_fc** to 1, so that the parser skeleton will enable the output parser to have function calls with an appropriate switch statement, instead of the usual function pointer array. This option is useful when you want to create parsers in Java or some other language that does not have function pointers. The default setting is 'fc=0' (no function calls).

'g' option: Grammar listing.

The 'g' option tells LRSTAR to print out a file with a complete grammar listing in a formatted style. The output file is named “?.grm.grammar.txt”.

'h' option: HTML grammar listing.

The 'h' option tells LRSTAR to print out the grammar in an HTML format. The option 'h=2' adds all EBNF generated symbols to the grammar listing.

'i' option: Ignore AST operations.

The 'i' option tells LRSTAR to ignore all AST operators and generate a parser which has no AST operations, (such as making nodes and calling node actions). This is useful if you have a grammar which is populated with AST operations and then you decide that you want to make a syntax checker (without the AST operations). You can get a syntax checker by using the 'i' option. Or perhaps you have a bug in your language processor and you want to find out if it is in one of your AST actions or in the parser. You can use the 'i' option to generate a parser that does not activate any parsing actions. If this parser has no bugs, then the bug is in your AST action, most likely. The default setting is 'i=0'.

'k' option: Keyword checking.

The 'k' option tells LRSTAR to do a keyword check when reading your grammar. That means, any alphabetical symbols (e.g. AbstractDeclarator, arguments, while, for, Statement) which are not defined as nonterminal symbols and not declared at the top of the grammar as keywords, will be flagged as errors. The 'k' option is for strict keyword checking. If you are going to use the 'k' option, you will have to declare all keywords at the top of the grammar, in order to generate a parser. Fortunately, LRSTAR will generate the list of all keywords it finds in your grammar, if you use the 'k' option. So, an easy way to get a list of all keywords is to use the 'k' option once and then cut and paste the list of generated keywords from the “?.grm.log.txt” file to your grammar file (?.grm). For keywords errors, LRSTAR will display a message something like this:

```
Calc.grm(56) :       Else  -> else Stmt...         *> else_
Calc.grm(56) : ---------------^
Calc.grm(56) : Keyword else was not declared.
```

'lalr' option: LALR(1) grammar analysis.

The 'lalr' option tells LRSTAR to do an LALR(1) grammar analysis. This is the default if no grammar analysis options are specified. There are three grammar analysis options: 'slr', 'lalr', and 'nd'. You may want to know if your grammar is SLR(1) category. Then you can use the 'slr' option and if there are no conflicts, then your grammar is SLR(1).

m' option: Minimize parser-table size.

The 'm' option tells LRSTAR to try several different orderings of the values in the parser tables and then generate the smallest parser tables, based on the data obtained. It will try 2 different orderings for the B matrix, 4 for the T matrix, 4 for the N matrix, and 2 for the R matrix. Usually, you can expect a 15% decrease in size by using the 'm' option. Here is a sample screen display when using the 'v' and 'm' option:

```
LRSTAR 4.0.000 Copyright 2013 Compilerware.

Input     DB2.grm /v /m

Grammar     1868 rules, 582 terminals, 643 nonterminals, 3687 tails.
States      3278 states before implementing shift-reduce actions.
Conflicts    145 states, 145 shift-reduce, 0 reduce-reduce.
Parser      1668 states after  implementing shift-reduce actions.

           ROWS   COLS         MATRIX      LIST     VECT      TOTAL
B matrix    447 x  382 x 1 =  170,754 -> 10,515 +  4,500 =   15,015
B matrix    447 x  382 x 1 =  170,754 -> 10,998 +  4,500 =   15,498
T matrix     80 x  382 x 2 =   61,120 -> 17,754 +  4,500 =   22,254
T matrix     80 x  382 x 2 =   61,120 -> 17,458 +  4,500 =   21,958
T matrix     78 x  289 x 2 =   45,084 -> 23,106 +  4,500 =   27,606
T matrix     78 x  289 x 2 =   45,084 -> 22,760 +  4,500 =   27,260
N matrix    307 x  378 x 2 =  232,092 -> 13,840 +  7,072 =   20,912
N matrix    307 x  378 x 2 =  232,092 -> 11,788 +  7,072 =   18,860
N matrix    307 x  275 x 2 =  168,850 -> 12,894 +  7,072 =   19,966
N matrix    307 x  275 x 2 =  168,850 -> 11,810 +  7,072 =   18,882
R matrix     50 x   80 x 2 =    7,782 ->  7,782 +  3,918 =   11,700
R matrix     50 x   80 x 2 =    7,782 ->  7,782 +  3,918 =   11,700

           ROWS   COLS         MATRIX      LIST     VECT      TOTAL
B matrix    447 x  382 x 1 =  170,754 -> 10,515 +  4,500 =   15,015
T matrix     80 x  382 x 2 =   61,120 -> 17,458 +  4,500 =   21,958
N matrix    307 x  378 x 2 =  232,092 -> 11,788 +  7,072 =   18,860
R matrix     50 x   80 x 2 =    7,782 ->  7,782 +  3,918 =   11,700
Total                                 -> 47,543 + 19,990 =   67,533

Skeleton: parser.cpp.skl
Output:   parser.cpp (5,985 lines)

0 min 1.125 sec, 290 conflicts, 0 warnings, 0 errors.
```

'na' option: Number of Arguments for each node.

The 'na' option tells LRSTAR how many node arguments you wish to have for each node name in the grammar. Each node name must have the same number of arguments, because the code in the parser handles each the creation of each node using the same code. This option will make sure that you don’t create an error by having 1 argument with one node name and 2 arguments with another node name.

'nd' option: Non-Determinism for reduce-reduce conflicts.

The 'nd' option tells LRSTAR to create non-deterministic actions for those Reduce-Reduce conflicts in the state machine.

'nd=2' option: Non-Determinism for all conflicts.

The 'nd=2' option tells LRSTAR to create non-deterministic actions for all the conflicts in the state machine, for both Reduce-Reduce and Shift-Reduce conflicts.

'o' option: Optimize parser-table speed.

The 'o' option tells LRSTAR to do remove do chain-reduction elimination. This gives in increase in parsing speed. However, in some grammars, there may not be any useless chain reductions. The default setting is 'o=0' (no optimization). Here is a grammar which has useless chain reductions:

```
Stmt   -> LeftSide '=' Exp ';'

Exp    -> Term
       -> Exp '+' Term
       -> Exp '-' Term

Term   -> Factor
       -> Term '*' Factor
       -> Term '/' Factor

Factor -> <number>
       -> <identifier>
       -> '(' Exp ')'
```

If the input line is, "y = 100;", the parser makes 3 reductions:

```
Factor <- <number>
Term   <- Factor
Exp    <- Term
```

After doing chain-reduction elimination, the parser only does one reduction as follows:

```
Exp <- <number>
```

This optimization may increase the size of the parser tables. I have seen cases where the parser-table size was twice as large after using the 'o' option. The increase in speed of your language processor may not be very much. You will have to experiment to find out.

'q' option: Quiet mode.

The 'q' option tells LRSTAR to suppress the display of most of the information written to the screen (your computer monitor). Here is an example of what LRSTAR shows on the screen when you use the 'q' option:

```
LRSTAR 4.0.000 Copyright 2013 Compilerware.
Input     Calc.grm ( exp s d=3 k q
0 min 0.015 sec, 0 conflicts, 0 warnings, 0 errors.
```

'rr' option: Reduce-Reduce conflicts only.

The 'rr' option tells LRSTAR to print only the reduce-reduce conflicts in the conflicts.txt file. The shift-reduce conflicts are usually not relevant and you don't want to see them. Since the default parsing action is the shift action, which is what you want, you don't need to examine them. However, the reduce-reduce conflicts are a serious problem and you need to resolve them somehow, whether by changing the grammar or writing a parse-action function to do a lookahead to resolve the problem. The default is 'rr=0' (show all conflicts, shift-reduce and reduce-reduce).

's' option: State machine listing.

The 's' option tells LRSTAR to print the state machine listing in the states.txt file. The default is 's=0' (no state-machine listing will be printed).

'slr' option: SLR(1) grammar analysis.

The 'slr' option tells LRSTAR to do an SLR(1) grammar analysis. There are three grammar analysis options: 'slr', 'lalr', and 'nd'. You may want to know if your grammar is SLR(1) category. Then you can use the 'slr' option and if there are no conflicts, then your grammar is SLR(1).

'tab=?' option: Tab setting for a grammar.

The 'tab=?' option tells LRSTAR how to treat the tab character when found in a grammar. Spacing is important in LRSTAR grammars. When defining a rule, the nonterminal (head) symbol being defined must appear before column 7. If you use 'tab=3' (the default setting) or 'tab=4', then everything should be fine and you can still put a tab before the nonterminal if you want. Putting this option in your grammar file, makes it portable (email to someone else and it still works without error, even if he uses a different tab setting in Visual Studio).

'v' option: Verbose mode.

The 'v' option tells LRSTAR to display more information on the screen that may be useful to you. At this time not much extra information is displayed, but in the future this may be more useful. The default setting is 'v=0'.

'w' option: Warning messages display.

The 'w' option tells LRSTAR to display the warning message on the screen. The default setting is 'w=0' (do not display the warning messages on the screen). Note: the warning messages are always displayed in the “?.grm.log.txt” file.

'wait' option: Wait for key press.

The 'wait' option tells LRSTAR 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).

Chapter 4.
Output Files.

LRSTAR 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.grm.conflicts.txt	A conflicts file, showing the conflicts in your grammar.
calc.grm.grammar.html	A grammar HTML file, showing your grammar.
calc.grm.grammar.txt	A grammar file, showing your grammar and the rule numbers.
calc.grm.log.txt	A log file, showing a log of what happened.
calc.grm.states.txt	A state-machine file, showing the parser state machine.

LRSTAR always attempts to create or modify the **.lgr** file, (in this case, the **calc.lgr** file). The **calc.lgr** is the lexical grammar file, input to DFASTAR. This **calc.lgr** file must have the correct terminal symbol numbers for the parser which was created. So LRSTAR will re-write the terminal symbol numbers in this **calc.lgr** file. If no **calc.lgr** file exists, then one will be created, but of course, it will not be complete, since it will only contain a list of the terminal symbols and not their definitions. You will have to add the definitions of some of the terminal symbols, such as <identifier>, before you can generate a lexer from this **calc.lgr** lexical grammar.

Of course, any pairs of (<skeleton> <output>) files will produce the appropriate <output> file. For example, the pair (**parser.cpp.skl parser.cpp**) will ask LRSTAR to create **parser.cpp** as output, and it will list all <output> files in the **calc.grm.log.txt** file.

Chapter 5.
Grammar Introduction

Definition

TBNF means Translational BNF. It is a BNF, Backus Normal Form, that completely defines the translation process from syntax to intermediate code (for compilers) or from syntax to processing functions (for interpreters). The paper, "A Translational BNF Grammar Notation (TBNF)", was published in SIGPLAN Notices in April 2006. This was the first publication to give a detailed explanation of this concept.

A Compiler Generator

Computer languages are necessary for programming computers. Compilers are needed to translate computer languages into machine code. Compilers are very difficult programs to write, so a program which builds a compiler automatically is a very useful thing, as long as it does not make the process too complicated. There are two differing opinions on this matter. Those who prefer to write compilers by hand and those who prefer to use a "compiler generator". So the real challenge is to build a "compiler generator" that makes the process easier and produces better compilers.

Compiler Parts

A compiler is most often composed of two separate programs: "front end" and "back end." TBNF pertains to the front end, which has 5 main parts: 1) lexer, 2) parser, 3) symbol-table, 4) abstract-syntax tree and 5) tree processor. Historically, BNF grammar notation only pertained to the syntax of a computer language. So the only part of a compiler that could be generated with BNF was the parser. However, TBNF grammar notation pertains to the parser, abstract-syntax tree and tree processing, so TBNF can define more of the compiler front end.

The Beginning

In 1984, Paul B Mann started writing an LALR parser generator in C. At that time there were not many parser generators available for the IBM PC/XT, only TWS by Frank DeRemer and Tom Pennello and QPARSER by Bill Barrett. Paul bought a TWS User's Manual for $25 and studied its BNF grammar notation and the QPARSER BNF notation. There was also the paper "A Human-Engineered Variant of BNF" by Henry Ledgard. Besides those, there was the ALGOL language definition and several others, which used an older style of BNF.

Different From YACC

By studying three or four sources, Paul created his own BNF. This was before YACC-like products were available for IBM PCs. So TBNF grammar notation evolved from a different origin than YACC. Paul thinks that turned out for the better. However, at the present time, YACC grammar notation is the dominant one in the compiler-development world. Almost everybody who creates a parser generator, wants to make it compatible with YACC so there will be no resistance to using it. That is fine if you just want a basic parser with code attached to the rules. However, if a parser generator is going to offer state-of-the-art features, it has to go beyond the YACC grammar notation and offer new symbolism. That's what you will find in TBNF, new symbolism for state-of-the-art concepts.

Modern Concepts In TBNF

1. Angled-bracket symbols to express variable symbols, such as <identifier>.
2. Symbol-table lookup specified in the grammar, such as <identifier> => lookup().
3. Curly-brace symbols to specify semantics, such as {typedef} in C language.
4. Extended BNF grammar notation, regular expressions in your grammar rules.
5. Automatically finding the keywords of your language, without forcing you to list all the keywords at the top of the grammar.
6. No code is allowed in your grammar, making your grammars clean and language independent.
7. Defined constants can be assigned to grammar symbols and used in your source code for better readability and reduced maintenance.
8. AST operators can be used to define the creation of an abstract-syntax tree automatically.
9. AST operators can be used to define node actions which will be called later when doing AST traversal.

LR vs LL Grammars

TBNF is LR grammar notation, which is preferable to LL grammar notation. LR grammars allow the freedom to use left and right recursion and specify operator precedence. LL grammars allow only right recursion, which is not natural. The following is an LR grammar segment showing left recursion in the symbol, ArgList:

```
ArgList   -> <identifier>
          -> ArgList ',' <identifier>
```

This is the way to specify the same thing in an LL grammar:

```
ArgList   -> <identifier>
          -> <identifier> ',' ArgList
```

Most people prefer the LR notation, and for this reason, LR is considered more natural than LL. However, many people do not like LR parsers because they are “bottom-up” in their recognition of the rules specified in the grammar, and this causes some problems when trying to produce output. LL parsers are “top-down” in their operation, so they have gained a big following.

Bottom-Up Processing With TBNF

The following TBNF grammar segment produces a “bottom-up” parser and “bottom-up” processing order for the functions specified (on the right side):

```
Start       -> Program... <eof>              => goal_
Program     -> program <id> '{' Stmt... '}'  => program_(2)

Stmt        ~> Target '=' Exp ';'            => store_

Target      -> <identifier>                  => target_(1)
Exp         -> Exp '+' Exp                   => add_
            -> Exp '-' Exp                   => sub_
            -> Exp '*' Exp                   => mul_
            -> Exp '/' Exp                   => div_
            -> <integer>                     => integer_(1)
            -> <identifier>                  => identifier_(1)
            -> '(' Exp ')'
```

Top-Down Processing With TBNF

The following TBNF grammar produces a a “bottom-up” parser, but a “top-down” processing order for the functions specified (on the right side):

```
Start       -> Program... <eof>            *> goal_
Program     -> program <id> '{' Stmt... '}'  *> program_(2)

Stmt        ~> Target '=' Exp ';'          *> store_

Target      -> <identifier>                *> target_(1)
Exp         -> Exp '+' Exp                 *> add_
            -> Exp '-' Exp                 *> sub_
            -> Exp '*' Exp                 *> mul_
            -> Exp '/' Exp                 *> div_
            -> <identifier>                *> identifier_(1)
            -> <integer>                   *> integer_(1)
            -> '(' Exp ')'
```

AST Operators (=> and *>)

How can this LR parser do “top-down” processing? The '=>' operator indicates “call this function at parsing time”. The '*>' operator indicates “make a node in the AST and call this function during traversal of the AST.” The LR parser does a “bottom-up” recognition of the input and builds the AST, but no processing is done until after the AST is completed. Then the AST traversal starts at the top and does a “top-down” processing. The AST traversal algorithm calls the node actions 3 times: 1) when going down the AST, 2) when passing over a node, and 3) when coming back from processing a nodes children. So what you end up with is a very versatile operation for creating output.

A Node Action

Here is what an “add_” node-action might look like:

```
short cNodeAction::add_( int n, int pass )
{
      switch( pass )
      {
         case TOP_DOWN:  break;
         case PASS_OVER: break;
         case BOTTOM_UP: emitstr( n, "\t\tADD\n" ); break;
      }
      return 0;
}
```

Chapter 6.
Grammar Symbols

Grammar Symbols

Grammar symbols are the “words” of your grammar which define the syntax of your computer language. Here is a list of the different types of symbols allowed in a 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_>

<semantic>

```
<semantic> -> '{' <alpha> '}'
```

Used as terminals, arguments.
Examples: {typedef}, {function_name}, {program_name}.

<amperalpha>

```
<amperalpha> -> '&' <alpha>
```

Used as arguments (refers to a node name).
Examples: &function_, &declaration_, &if_stmt.

<literal>

```
<literal>    -> ''' LChar+ '''
LChar        -> '\' '''
             -> '\' '"'
             -> '\' '\'
             -> '\' letter
             -> AnyLChar
AnyLChar     -> 0..255 - ''' - '"' - '\' - ' ' - \n - \z
```

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

<string>

```
<string>      -> '”' '”'
              -> '”' SChar+ '”'
SChar         -> '\' '''
              -> '\' '"'
              -> '\' '\'
              -> '\' letter
              -> AnySChar
AnySChar      -> 0..255 - ''' - '"' - '\' - \n - \z
```

Used as arguments only.
Examples: “hello world\n”, “&0:\tif\n”, “integer”, “float”.

<integer>
<integer> -> digit+

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

Reserved Grammar Symbols

Only three grammar symbols are reserved and have a special meaning:

<error>

Used as a terminal, for an input symbol that causes a syntax error.
Examples: <error>, [<error>]...

<eof>

Used as an end-of-file terminal symbol in the first rule of a grammar.
Examples: Start -> Statements <eof>

<keyword>

Used as a terminal, to represent any keyword of the language (any keyword that is not reserved).
Examples: <keyword>, [<identifier>|<keyword>]/','...

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 “is a” (starts the right side of a rule, same as ‘->’).
'->'	means “is a” (starts the right side of a rule).
'~>'	means “is a” (same as '->', but reverse the order in the AST).
'/->'	means “is a” (same as '->', but choose this rule in case of conflict).
'/~>'	means “is a“ (same as '~>', but choose this rule in case of conflict).
'|'	means “alternate choice” for another rule, (e.g. Color -> red | white | blue).
';'	means “end of a group of rules for a nonterminal” (';' is optional).

Action Operators

Action operators are placed at the end of a rule. Only one action operator may be placed at the end of a rule, except for the combination of (‘=>’ and ‘+>’) or (‘=>’ and ‘*>’).

'=>'	means “parse action” (call this function at parsing time).
'+>'	means “make a node” (make a node in the AST).
'*>'	means “make a node” and “call a node action” (during AST traverrsal).
'=+>'	means “parse action” and “make a node”.
'=*>'	means “parse action”, “make a node” and “call a node action”.

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)).
'/'	means “one or more occurrences, separated by”, (e.g. Arg/','...)
'~'	means “reverse the order in the AST”, (e.g. (a b c)~).
'~..'	means the same as **'...'** but reverse the order in the AST. (e.g. (a | b | c)~..).

Operator Precedence Notation

'{'	means “operator precedence start”.
'}'	means “operator precedence end” , (e.g. { '+' '-' }).
'<<'	means “left associative”, used after a list of operators, (e.g. { '+' '-' } <<).
'>>'	means “right associative”, used after a list of operators, (e.g. { '^' } >>).

Reserved Word Designator

'^' means “reserved word”, placed after a keyword declaration, (e.g. if^, else^).

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
*/
```

Nonterminal Symbols

Nonterminal symbols are the those symbols which are defined to be a sequence of other symbols. Nonterminals may be composed of letters, numbers and underscores ('_'). The first character cannot be a number. Here is a list of valid nonterminal symbols:

```
Start, IfStatement, For_loop, Expression1, Expression_2
```

Terminal Symbols

Terminal symbols are those symbols of the grammar that are pre-defined, such as keywords, identifiers, numbers, operators, punctuators, character strings, etc. Most of these symbols are defined in the lexical grammar and recognized by the lexer. It is this separation of the parser and lexer that allows one to define a grammar without lots of ambiguity. The concept of a separate lexer also presents a coherent set of primitive symbols to someone learning the language.

```
<identifier>, '=', for, while, '&&', <eof>, <string>, {typedef}, {function_name}.
```

Keywords

Keywords are composed of letters, numbers and underscores ('_'). The first character must be a letter. Keywords may be specified as literal symbols using single quotes (e.g. 'while'), but this is not necessary unless they contain some special character (e.g. '<html>'). Here is a sample showing some keywords:

```
Type         -> char | int | short | unsigned

MetaStart    -> '<META'
MetaItem     -> 'HTTP-EQUIV' '=' Value
             ->  NAME        '=' Value
             ->  CONTENT     '=' Value
```

Literal Symbols

Literal symbols are symbols that begin and end with a single quote ('). Any character can be specified within single quotes, except end-of-line, tab, space, single quote (‘), double quote (“) and backslash (\). To have these unallowed characters in a literal, they must be preceded by a backslash (\), for example: '\n', '\t', '\'', '\”', '\\', etc. Here is an example:

```
PostfixExp  -> PrimaryExp
            -> PostfixExp '[' Exp ']'
            -> PostfixExp '(' [Exp] ')'
            -> PostfixExp '->' Name
```

Lexical Symbols

Lexical symbols are those symbols of the grammar such as <identifier>, <number>, <string>, etc. These have no literal form. For example, all words made up of letters, such as WinMain, CloseFile, MsgBox, etc. are usually represented in a grammar by <identifier>. These are words recognized by the Lexer. Lexical symbols begin with '<' and end with '>'. Inside the angle brackets, you may use letters, numbers and underscores. Here are some examples:

```
Parameter   -> <identifier>
            -> <integer>
            -> <hexdigits>
            -> <string>
```

Semantic Symbols

Semantic symbols begin with '{' and end with '}', (e.g. {typedef}). They are variable symbols of the grammar that are defined by the lexer to be one terminal (e.g. <identifier>) and need to be changed later into another terminal (e.g. {typedef}). For example, a function name, “main”, is first recognized to be an <identifier>, but later it may need to become a {function_name}. Another classical case is an <identifier> that is defined to be a 'typedef' in the C language. Here is a grammar segment that solves the 'typedef' problem:

```
<identifier>  => lookup ()

Declaration   ->         VarDecl1 /','... ';'
              -> typedef VarDecl2 /','... ';'

VarDecl1      -> Type... <identifier>
VarDecl2      -> Type... <identifier>  => defterm (2, {typedef})

Type          -> SimpleType...
              -> Type '*'

SimpleType    -> char
              -> int
              -> short
              -> unsigned
              -> {typedef}
```

Here is a difficult input statement that will be parsed correctly without any errors:

```
typedef unsigned int uint;  uint *uintptr;
```

In this line, when 'uint' is encountered it will be recognized as an <identifier> by the lexer. At the comma (,) ‘uint’ will be changed to a {typedef} by the parse action, 'defterm'. At the second encounter with 'uint', it will be recognized as a {typedef} by the symbol-table lookup function. Therefore the parsing is handled correctly.

Chapter 7.
Grammar Sections

TBNF Sections

TBNF grammars have 4 sections:

1. Defined Constants (optional),
2. Token Declarations (optional).
3. Operator Precedence (optional).
4. Syntax Rules.

1. Defined Constants (optional)

Defined constants can be defined at the top of the grammar. They can have a numeric value or be associated with terminal symbols. These will be useful ins your grammar as arguments to parsing actions and useful in your source code as well. Let's take a look at some examples:

```
// DEFINED CONSTANTS:
NUMBER_OF_KEYWORDS  36;
VERSION             3;
BUILD               138;
IDENTIFIER          <identifier> => lookup()
STRING              <string>;
INTEGER             <integer>;
FLOAT               <float>;
```

2. Token Declarations (optional)

Token declarations are specified at the top of a grammar. Any processing that needs to be done is specified here. '=>' indicates "call this function". For example, all <identifier>s need to be looked up in the symbol table. The symbol-table index number is put on the parse stack and later transferred to a node in the AST. Here is an example:

```
// TOKENS GOING INTO THE SYMBOL TABLE
            <identifier>  => lookup()
            <string>      => lookup()
INTEGER     <integer>     => lookup()
FLOAT       <float>       => lookup()
```

In the above example, <identifier>, <integer> and <string> are input symbols that we want to be put into the symbol table by calling the function, lookup(). We only have to list those tokens which need to be processed.

3. Operator Precedence (optional)

The Operator Precedence section allows one to define which operators have priority over others. For example, when we are parsing expressions, such as 1+2*3, we want the 2*3 to be calculated first. We specify this operator precedence as follows:

```
{ '+' '-' } <<   // lower priority
{ '*' '/' } <<   // higher priority
```

The operators with higher priority are listed below ones with lower priority. The ‘<<’ means left associative and ‘>>’ means right associative. A case of right associativity would be the exponential operator ‘^’, for example 2^2^3 would be calculated as 2^8, instead of 4^3. We want the 2^3 to be computed first. This operator usually has higher priority than the ‘*’ and ‘/’ operators. The operator precedence would look like this:

```
{ '+' '-' } <<  // left  associative
{ '*' '/' } <<  // left  associative
{ '^'     } >>  // right associative
```

4. Syntax Rules

A rule is a nonterminal symbol followed by a '->' and zero or more symbols. Any null production (empty rule) must be listed first. Rules have this format:

```
Nonterminal ('->' GrammarSymbols )...

GrammarSymbols -> (Nonterminal | Terminal)...
```

Start Symbol or Goal Symbol

The first rule of a grammar must define the start symbol. The right side of the start production contains one symbol and the end-of-file symbol (e.g. <eof>). The end-of-file symbol cannot be used anywhere else in the syntax-rules section grammar. Here are some examples of start definitions:

```
Start -> Input <eof>

Goal  -> [CompilationUnit]... <eof>

Input -> Statements... <eof>
```

Nonterminals

A nonterminal symbol definition may contain many rules. Here is an example taken from a C grammar:

```
Stmt  -> ';'
      -> AssignmentExp ';'
      -> goto Identifier ';'
      -> return [Exp] ';'
      -> if '(' Exp ')' Stmt
      -> if '(' Exp ')' Stmt else Stmt
      -> switch '(' Exp ')' '{' Cases Default '}'
      -> while '(' Exp ')' Stmt
      -> do Stmt while '(' Exp ')' ';'
      -> for '(' Exp1 ';' Exp2 ';' Exp3 ')' Stmt
```

As you can see, the only special symbol you need for rules is the '->' which means "is a" and is placed at the beginning of each rule. '|' is allowed for defining a subsequent rule in a grammar.

EBNF Notation

EBNF notation is accepted by LRSTAR. These are the regular expression operators allowed:

```
|           means "or" (an alternate choice).
...         means "repeating" (one or more occurrences).

(           means "group start".
)           means "group end".
)...        means "group end, one or more occurrences".

[           means "optional start".
]           means "optional end".
]...        means "optional group end, zero or more occurrences".

/x...       means "list separated by x".

)~          means "group end", reverse order in AST.
]~          means "optional group end", reverse order in AST.
)~..        means "group end, one or more occurrences", reverse order in AST.
]~..        means "optional group end, zero or more occurrences", reverse order.
/x~..       means "list separated by x", reverse order in AST.
```

Here are some combinations of EBNF that are useful:

```
(A|B|C)        means A or B or C.
(A|B|C)...     means one or more occurrence of A or B or C.
[A|B|C]...     means zero or more occurrences of A or B or C.

[x]...         means zero or more occurrences of x.
[x...]         means zero or more occurrences of x (same as above).

Color/','...   means a comma separated list of Colors.
(X|Y|Z)/';'... means a list of X or Y or Z separated by semi-colons.
[X|Y|Z]/';'... means an optional list separated by semi-colons.

(A|B|C)~..     means one or more occurrence of A or B or C, reverse order in AST.
[A|B|C]~..     means zero or more occurrences of A or B or C, reverse order in AST.

(X|Y|Z)/';'~.. means a list of X or Y or Z separated by semi-colons, reverse order.
[X|Y|Z]/';'~.. means an optional list separated by semi-colons, reverse order.
```

Here is a sample taken from an SQL grammar:

```
SelectClause -> SELECT [ALL | DIST] SelectList FromClause
                [Where | Groupby | Having | Intersect | Orderby]...

Where         -> WHERE SearchCond
Groupby       -> GROUP BY ColumnList
ColumnList    -> ColumnSpec /','...
Having        -> HAVING SearchCond
Intersect     -> INTERSECT StmtName
              -> MINUS     StmtName
              -> UNION     StmtName
              -> UNION ALL StmtName
Orderby       -> ORDER BY ColumnSpecList
              -> FOR UPDATE OF ColList
```

Actions

Any rule may have an action attached to it (at the end of the rule). A rule may have 1 or 2 actions. If 2 actions, the first action must be ‘=>’ and the second action may be either '+>' or '*>'. There are 5 types of actions:

=>	Call a parse-action.
+>	Make a node in the AST
*>	Make a node and call a node-action (during AST traversal).
=+>	Call a parse-action and make a node in the AST.
=*>	Call a parse-action, make a node and call a node-action.

=> Parse action

Parse actions are done by the parser when a rule is recognized. When the last symbol in a rule is accepted, the parser makes a reduction and calls the function indicated by the '=>' operator. Here is the syntax:

```
'=>' FunctionName '(' Arg/','... ')'

Arg may be any one of:

Terminal symbol         (e.g. <identifier>)
Integer                 (e.g. 100)
Defined Constant        (e.g. INTEGER)
Literal                 (e.g. ':')
String                  (e.g. “hello world\n”)
NodeName                (e.g. &program_)

NodeNames are indicated by an '&' character preceeding the node name.
```

Here is an example:

```
Program -> program <identifier> '{' Stmts '}' => defterm (2, {program})
```

What happens here? The supplied function, defterm(), will be called and it will redefine the terminal number for the <identifier> to be {program} in the symbol table. The next time the parser sees this <identifier>, it will be classified as a {program} by the symbol-table lookup function. Consider this input line:

```
program bubblesort
{
   /* Empty function. */
}
```

In the above input line, the defterm() function will define the terminal type for 'bubblesort' to be {program}. The next occurrence of 'bubblesort' will be treated by the parser as a {program} and not an <identifier>. This is a very useful operation. This typing operation solves many ambiguity problems in computer languages. This is not hacking, like some authors claim. It is a natural thing we all do when reading languages, use our memory. It is far better to make use of memory (symbol table), than try to live with ambiguity in a grammar.

+> Make a node in the AST

Make-a-node operation occurs when the parser makes a reduction of a rule. At this time, the parser creates a new parent node and attaches any children nodes to it as may occur in the rule being recognized. Here is an example of how to do expressions, taken from the 'calc.grm':

```
Exp      -> Primary
         -> Exp '+' Exp        +> add
         -> Exp '-' Exp        +> sub
         -> Exp '*' Exp        +> mul
         -> Exp '/' Exp        +> div
```

Here is the syntax for specifying three node arguments to take advantage of the code already in the parser.cpp.skl. Actually you may change the parser.cpp.skl to do whatever you want and have as many arguments as you want, as long as you specify the option that indicates how many arguments you want (e.g. na=3).

```
+> NodeName
+> NodeName '(' ')'
+> NodeName '(' Index ')'
+> NodeName '(' [Index] ',' Terminal ')'
+> NodeName '(' [Index] ',' [Terminal] ',' Type ')'
```

NodeName

NodeName is the name of the node being created. NodeNames can be composed of letters, numbers and underscores ('_'). The first character cannot be a number.

Index

Index indicates which symbol in the grammar rule is to be associated with the new node. This is a number indicating the relative position in the right side of the grammar rule (e.g. 1, 2, 3). It is usually desirable to put all variable symbols, such as <identifier>, into a node for later processing during AST traversal. Therefore all these terminal symbols should have a node name specified along with an Index number.

Terminal

Terminal indicates “change the terminal number to”. This handles the classical “typedef” problem. It is also useful for other things, like function names that have been declared previously that need to be distinguished from variable names or array names. A typical grammar rule would be like this:

```
Declaration -> typedef TypeList <identifier> ';'  +> typedefdecl (3,{typedef})
```

Type

Type indicates a type (most likely a defined constant, such as INT or FLOAT) which will be assigned to this symbol in the symbol table. This type will be useful when later processing the AST to do type analysis. Here is an example:

```
Declaration -> int   <identifier> ';'  +> decl_ (2,,INT)
            -> float <identifier> ';'  +> decl_ (2,,FLOAT)
```

*> Make a node and call a node action

A Make-a-node operation occurs when the parser makes a reduction of a rule. At this time, the parser creates a new node and attaches the children nodes found in the right side of the rule. Here is the syntax:

```
*> NodeName                                          NodeAction ['(' Args ')']
*> NodeName '(' ')'                                  NodeAction ['(' Args ')']
*> NodeName '(' Index ')'                            NodeAction ['(' Args ')']
*> NodeName '(' [Index] ',' Terminal ')'             NodeAction ['(' Args ')']
*> NodeName '(' [Index] ',' [Terminal] ',' Type ')'  NodeAction ['(' Args ')']
```

NodeAction name needs to be specified.

Args may be specified, as many as you want.

Arg may be any one of:

Terminal symbol	(e.g. <identifier>)
Integer	(e.g. 100)
Defined Constant	(e.g. INTEGER)
Literal	(e.g. ':')
String	(e.g. “hello world\n”)
NodeName	(e.g. &program_)

NodeNames are indicated by an '&' character preceeding the node name.

=+> Call a parse action and make a node

This is the combination of two operations, which is more concise than having a => and an +> for the rule. In this case (=+>) the parse action name will be the same as the node name and no arguments will be specified for the parse action. Some people have found this to be useful for their application.

=*> Call a parse action , make a node, and call a node action

This is the combination of three operations. Some people may need this kind of thing. The parse action name will be the same as the node name and the node action name will be the same as the node name also, unless a node action name is specified after the node name. Here are two different possibilities:

```
Assignment  ~> Target  '=' Exp ';'   =*> assignment

Assignment  ~> Target  '=' Exp ';'   =*> assignment  emit (,,”store”)
```

In the first case we will need a parse action called assignment(), and a node action called assignment(). In the second case we will need a parse action called assignment() and a node action called emit().

Chapter 8.
Grammar Examples

Example 1. A Simple Expression Grammar

Consider this simple expression grammar:

```
{ '+'  '-' } <<
{ '*'  '/' } <<

Start    -> Stmt... <eof>

Stmt     -> Target '=' Exp ';'

Target   -> <identifier>

Exp      -> Primary
         -> Exp '+' Exp
         -> Exp '-' Exp
         -> Exp '*' Exp
         -> Exp '/' Exp

Primary  -> <identifier>
         -> <integer>
         -> '(' Exp ')'
```

This grammar describes a syntax but says nothing about the creation of an AST. We attach to the end of certain rules a notation as follows:

```
+> node_name
```

Which means "when this rule is recognized, make a node in the AST with this name". And if the LR parser attaches the nodes to the AST, we will get a complete tree with a root node, intermediate nodes and leaf nodes, just like we want. Here is the improved grammar:

```
{ '+'  '-' } <<
{ '*'  '/' } <<

Start    -> Stmt... <eof>

Stmt     -> Target '=' Exp ';'    +> store

Target   -> <identifier>          +> target(1)

Exp      -> Primary
         -> Exp '+' Exp           +> add
         -> Exp '-' Exp           +> sub
         -> Exp '*' Exp           +> mul
         -> Exp '/' Exp           +> div

Primary  -> <identifier>          +> ident(1)
         -> <integer>             +> int(1)
         -> '(' Exp ')'
```

Note: the (1) notation means, "Attach the value found in the parse stack at relative position one to the new node being added to the AST." We need to have these values in the AST. Actually, it is not a value, it is a symbol-table index which points to the symbol where the value resides.

So far this is good, however, this notation is lacking one thing. The resultant AST will reflect the order of the input source code, which may not be the order we want when traversing the AST. In this case the AST order created from the rule:

```
Stmt -> Target '=' Exp ';'  +> store
```

Will look like this:

```
+ store
  + Target
  + Exp
```

The problem with this is that the tree traversal process would normally traverse the Target first and the Exp second. This is the opposite of what we want when generating intermediate code. This is the order we really want in the AST:

```
+ store
  + Exp
  + Target
```

This allows the traversal to generate code for the Exp before generating the code for storing the result in the Target. Another symbol is needed here to indicate, "Reverse the order of nodes for this rule, when adding these nodes to the AST." We use the symbol (~>) instead of the (->) for this as follows:

```
Stmt ~> Target '=' Exp ';'  +> store
```

Now we have a notation which can automate the second phase of a language processor, the construction of the AST in the correct order. When given this input source line:

```
x = (1+2/a) * (3-4/b);
```

Our generated parser will build an AST that looks like this:

```
+ root
  + store
    + mul
    | + add
    | | + int (1)
    | | + div
    | |   + int (2)
    | |   + ident (a)
    | + sub
    |   + int (3)
    |   + div
    |     + int (4)
    |     + ident (b)
    + target (x)
```

Creating Instruction Codes From the AST

Now comes the third phase, creating output codes from the AST. We need a notation in the grammar attached to each node name which indicates the instruction we want to emit and the values for the computations. Consider that each node in the AST will be traversed at least two times, once when going down the tree and once when going back up the tree. Some nodes will experience an intermediate "pass over" traversal. These are the parent nodes when a parent has more than one child. For example, consider an argument list: (a, b, c), and its AST:

```
+ arg_list
  + arg (a)
  + arg (b)
  + arg (c)
```

If we think of a “pass over” traversal that passes over the parent node (arg_list) when going from one child to the next, then we have two passes over the “arg_list” node, one when going from (a) to (b) and another when going from (b) to (c). If we have a notation to generate a “(“ at the top-down pass, a “,” at the “pass over” pass, and a “)” at the bottom-up pass, when we traverse the AST we will get this output:

```
(a,b,c)
```

We could represent this in the grammar notation with the following:

```
ArgList -> ‘(‘ Arg/‘,‘... ‘)‘     *> arg_list     emit ( “(“, “,“, “)“ )
Arg     -> <identifier>           *> arg (1)      emit ( “%s“ )
```

During top-down, emit “(“. During pass over, emit “,”. During bottom-up, emit “)”. During the bottom-up pass, emit as a string the value attached to the node (a, b, or c). Take a look at the complete TBNF grammar for the simple expression grammar:

```
{ '+'  '-' } <<
{ '*'  '/' } <<

Start    -> Stmt... <eof>

Stmt     ~> Target '=' Exp ';'    *> store        emit (,,"STORE\n")

Target   -> <identifier>          *> target(1)    emit (,,"LADR %s\n")

Exp      -> Primary
         -> Exp '+' Exp           *> add          emit (,,"ADD\n")
         -> Exp '-' Exp           *> sub          emit (,,"SUB\n")
         -> Exp '*' Exp           *> mul          emit (,,"MUL\n")
         -> Exp '/' Exp           *> div          emit (,,"DIV\n")

Primary  -> <identifier>          *> ident(1)     emit (,,"LOAD %s\n")
         -> <integer>             *> int(1)       emit (,,"LOAD %s\n")
         -> '(' Exp ')'
```

When we process the input statement below:

```
x = (1+2/a) * (3-4/b);
```

We get the same AST as shown above:

```
+ root
  + store
    + mul
    | + add
    | | + int (1)
    | | + div
    | |   + int (2)
    | |   + ident (a)
    | + sub
    |   + int (3)
    |   + div
    |     + int (4)
    |     + ident (b)
    + target (x)
```

And this intermediate code:

```
LOAD 1
LOAD 2
LOAD a
DIV
ADD
LOAD 3
LOAD 4
LOAD b
DIV
SUB
MUL
LADR x
STORE
```

This is exactly what we want. This output is suitable for a stack based interpreter, which could execute these instructions directly or generate object code.

Example 2. A calculator with if statements

```
/* Input Tokens. */

<error>     => error()  // call error handler.
<ident>     => lookup() // symbol-table lookup.
<integer>   => lookup() // symbol-table lookup.

/* Operator precedence. */

{ '==' '!=' }  <<
{ '+'  '-'  }  <<
{ '*'  '/'  }  <<

/* Rules. */

Start -> Pgm... <eof>

Pgm   -> program <ident> '{' Stmt... '}' *> program(2) emit("PROGRAM %s\n",,"END\n")

Stmt  ~> Target '=' Exp ';'          *> store     emit(,,"STORE\n")
      -> if RelExp Then endif        *> if        emit("if&0:\n",,"endif&0:\n")
      -> if RelExp Else endif        *> if        emit("if&0:\n",,"endif&0:\n")
      -> if RelExp Then2 Else2 endif *> if        emit("if&0:\n",,"endif&0:\n")

Then   -> then Stmt...               *> then      emit("BR NZ,endif&1\nthen&1:\n",,)
Else   -> else Stmt...               *> else      emit("BR Z,endif&1 \nelse&1:\n",,)
Then2  -> then Stmt...               *> then      emit("BR NZ,else&1 \nthen&1:\n",,)
Else2  -> else Stmt...               *> else      emit("BR endif&1   \nelse&1:\n",,)
Target -> <ident>                    *> target(1) emit(,,"LADR %s\n")

Exp    -> Primary
       -> Exp '+' Exp                *> add       emit (,,"ADD\n")
       -> Exp '-' Exp                *> sub       emit (,,"SUB\n")
       -> Exp '*' Exp                *> mul       emit (,,"MUL\n")
       -> Exp '/' Exp                *> div       emit (,,"DIV\n")

RelExp -> Exp '==' Exp               *> eq        emit (,,"EQ\n")
       -> Exp '!=' Exp               *> ne        emit (,,"NE\n")

Primary -> <ident>                   *> ident(1)  emit (,,"LOAD %s\n")
        -> <integer>                 *> int(1)    emit (,,"LOAD %s\n")
        -> '(' Exp ')'
```

Sample Source Code Input

```
program test
{
     if a == 0
     then
        if x == 0
        then b = 10;
        else b = 20;
        endif
     else
        if x == 1
        then b = 30;
        else b = 40;
        endif
     endif
}
```

AST Constructed by the Parser

```
+ root
  + program (test)
    + if
      + eq
      | + ident (a)
      | + int (0)
      + then
      | + if
      |   + eq
      |   | + ident (x)
      |   | + int (0)
      |   + then
      |   | + store
      |   |   + int (10)
      |   |   + target (b)
      |   + else
      |     + store
      |       + int (20)
      |       + target (b)
      + else
        + if
          + eq
          | + ident (x)
          | + int (1)
          + then
          | + store
          |   + int (30)
          |   + target (b)
          + else
            + store
              + int (40)
              + target (b)
```

Intermediate Code Output from the AST Traversal

```
          PROGRAM test
if1:
          LOAD a
          LOAD 0
          EQ
          BR NZ,else1
then1:
if2:
          LOAD x
          LOAD 0
          EQ
          BR NZ,else2
then2:
          LOAD 10
          LADR b
          STORE
          BR endif2
else2:
          LOAD 20
          LADR b
          STORE
endif2:
          BR endif1
else1:
if3:
          LOAD x
          LOAD 1
          EQ
          BR NZ,else3
then3:
          LOAD 30
          LADR b
          STORE
          BR endif3
else3:
          LOAD 40
          LADR b
          STORE
endif3:
endif1:
          END
```

Example 3. Delayed Typing Of <identifier>s

There are some occasions in which it is very useful to delay the typing of symbols until the whole rule is recognized. This allows one to avoid conflicts in the grammar or avoid writing code that does a look-ahead to resolve a conflict at parsing time. Here is a classical example often presented as a reason why you need infinite look-ahead to resolve this ambiguity in the C language:

```
ExternalDef  -> FuncDecl
             -> FuncDef

FuncDecl     -> Type FuncDeclName '(' Arg/','... ')' ';'
FuncDef      -> Type FuncDefName  '(' Arg/','... ')' '{' Stmts '}'

FuncDeclName -> <identifier>               => funcdecl_(1)
FuncDefName  -> <identifier>               => funcdef_(1)
```

In the above grammar there is a reduce-reduce conflict. At the input symbol '(' the parser cannot decide whether <identifier> is a FuncDeclName or a FuncDefName. In fact, it cannot make this decision until it sees the ';' or the '{'. We can get rid of the ambiguity (conflict) by re-writing the grammar this way:

```
ExternalDef  -> FuncDecl
             -> FuncDef

FuncDecl     -> Type <identifier> '(' Arg/','... ')' ';'
FuncDef      -> Type <identifier> '(' Arg/','... ')' '{' Stmts '}'
```

In this new grammar, there is no conflict, however, there is no parse actions for either case. We must delay the action and perform it later during traversal of the AST. This is not a problem, however, it may seem foreign to you. You can learn to do it this way. I think it's better than doing a look-ahead for the ';' or the '{'. We need to incorporate the appropriate node actions in the grammar as follows:

```
ExternalDef  -> FuncDecl
             -> FuncDef

FuncDecl     -> Type <identifier> '(' Arg/','... ')' ';'             *> funcdecl_(2)
FuncDef      -> Type <identifier> '(' Arg/','... ')' '{' Stmts '}' *> funcdef_(2)
```

Now, we will have two different nodes in the AST, one called 'funcdecl_' and another called, 'funcdef_'. The <identifier> will be attached to the node via its symbol-table-index. At this time we can do appropriate processing of the names. This method avoids the look-ahead problem.