Cocktail FAQ
Frequently Asked Questions

J. Grosch

DR. JOSEF GROSCH

COCOLAB - DATENVERARBEITUNG

GERMANY

Cocktail

Toolbox for Compiler Construction

Cocktail FAQ - Frequently Asked Questions

Josef Grosch

Oct. 2, 1998

Document No. 35

Dr. Josef Grosch
CoCoLab - Datenverarbeitung
Breslauer Str. 64c
76139 Karlsruhe
Germany

Phone: +49-721-91537544
Fax: +49-721-91537543
Email: grosch@cocolab.com

Cocktail FAQ - Frequently Asked Questions

Josef Grosch
CoCoLab
Hagsfelder Allee 16
D-76131 Karlsruhe
Germany

Tel: +49-721-697061
Fax: +49-721-661966
EMail: grosch@cocolab.de

Abstract

This list answers some frequently asked questions about the Cocktail Toolbox. Although most of the questions are about efficiency there are some other topics as well.

1. What does COCKTAIL stand for?

Cocktail stands for

> COmpiler Compiler ToolkIt KArLsruhe or
> COmpiler CompIler Toolbox KArLsruhe.

Some creativity is needed to bring the relevant characters into the right order.

2. How to generate efficient compilers?

When we want to construct efficient software, we want to reduce or optimize two aspects: run time and storage. There are measures that improve both aspects together and there are measures that improve one aspect at the expense of the other.

The Cocktail Tools have been designed originally for the operating system UNIX which usually provides a sufficient amount of storage. Therefore the compiler parts generated by Cocktail are primarily oriented towards run time efficiency which sometimes causes an increased need for storage.

When we are asking for an efficient compiler we have to specify first what we want to reduce primarily: run time or storage. Then we have to construct all compiler parts according to this goal: input handling, scanner, parser, etc. .

2.1. How to make input as efficient as possible?

Input is handled in the module *Source* by the procedure *GetLine*. The default version of this module reads large blocks of characters from a file. This is actually done using the procedure *rRead* from the module *rSystem* in the reuse library. The only concern is the aspect of run time.

When reading from a file using rRead, select a fast variant of rRead from the following possibilities (in order of decreasing speed):

- Use the UNIX system call read.
- Use the function fread of the C library.
- Use the function fgets of the C library.

General hints:

- Let the procedure GetLine return as large blocks of characters as possible up to the maximal size given by the argument Size.
- Avoid any preprocessing such as e. g. truncating line length to 72 characters.

2.2. How to generate efficient scanners?

2.2.1. How to generate fast scanners?

- Specify separate patterns for the keywords.
- Avoid backtracking (or backing up) by following the next hints (in decreasing importance):

. Avoid the use of right context where both regular expressions match an arbitrary number of characters.

. Avoid the use of right context where one regular expression matches a fixed number of characters.

. Avoid the use of right context at all.

. Avoid the use of the action yyLess.

- The predefined rule for skipping blanks is faster than an explicitly specified rule. Therefore, usually it is a good idea to avoid rules such as

```
" "       :- {}
" " +     :- {}
```

- Avoid the use of the left justification operator <.
- Compute source positions only where necessary. This means to introduce the action part in rules with :- instead of :.
- Avoid unnecessary copying of tokens by calling GetWord and so on.
-

Replace	l = GetWord (s); x = MakeIdent (s, l);
by	x = MakeIdent (TokenPtr, TokenLength);
Replace	l = GetWord (s); x = PutString (s, l);
by	x = PutString (TokenPtr, TokenLength);
Replace	GetWord (s); x = atoi (s);
by	x = atoi (TokenPtr); if possible

- Avoid calling of procedures in the actions, prefer inline code.
- Avoid wrapper functions such as:

```
GLOBAL {
   int GetToken ARGS ((void))
   {
      register int Token = Get_Token ();
      ...
      return Token;
   }

   # define GetToken Get_Token
}
```

2.2.2. How to generate small scanners?

A generated scanner uses an initial buffer for 8448 characters. It also contains a stack of states of that size (16896 bytes) which yields an initial space requirement for the main data structures of 25344 bytes.

- Decrease the initial buffer size by using a #define directive such as:

```
# define yyInitBufferSize       1024 * 1 + 256
```

- Decrease the size of the stack for include files by using a #define directive such as:

```
# define yyInitFileStackSize    4
```

- Do not specify separate patterns for the keywords. Recognize keywords with the pattern for identifiers and distinguish both cases with an additional mechanism. Might be worth while if there are many keywords - and the additional mechanism also uses space.
- Minimize the number of start states.
- Avoid the constructs for counted repetition of regular expressions such as e. g.

```
a [3]
a [0-4]
```

  In most cases this causes a big increase in table size.
- Minimize the maximal length of tokens, because otherwise the buffer size could increase dynamically. Example: Do not recognize comments as one token but break it up into smaller parts.
- Avoid include files, that means avoid calls of BeginFile, because this will allocate additional buffers.
- Use options -r and -o, do not use option -i, if possible.

-o	optimize table size
-r	reduce number of generated switch labels
-i	use ISO 8 bit code instead of ASCII 7 bit code

- Unused functions should be removed from the generated scanner such as e. g. GetWord, GetUpper, GetLower, input, yyPush, yyPop, etc. (This is done automatically.)
- Unused arrays should be removed from the generated scanner such as e. g. yyToLower, yyToUpper, and yyStStStack. (This is done automatically.)

2.3. How to compute the end position of tokens?

The line and column numbers of the last character of all tokens can be computed by including a "wrapper" procedure into the GLOBAL section similar to the following:

```
int GetToken (void)
{
   int Token = Get_Token ();

   if (Token != EofToken) {
      Line   = yyLineCount;
      Column = (unsigned char *) TokenPtr - yyLineStart + TokenLength - 1;
   }
   return Token;
}

# define GetToken Get_Token
```

This code uses the two internal variables yyLineCount and yyLineStart which are not documented, otherwise. It works even in case of tokens that are spread over several lines or recognized by a series of rules.

2.4. How to handle include files?

The scanning of an include file is triggered by a call of the procedure BeginFile with the name of the include file as argument. This procedure can be called from the scanner or from the parser. In both cases, care has to be taken of the exact moment of the call. What BeginFile does is that it inserts the contents of the include file after a certain character of the current file. This character is the one that has been recognized last by the scanner.

2.4.1. How to handle include files in the scanner?

Let’s assume the scanner specification contains the following rule:

```
"#" INCLUDE identifier   : { char word [256];
                             GetWord (word);
                             BeginFile (word + 8);
                             push (word + 8);
                             return ...;
                           }
```

This rule does not care about white space that might surround the keyword INCLUDE. The contents of the include file is inserted exactly after the last character of the identifier. This is done independently of any lookahead that the scanner might need in order to recognize this rule. In case of lookahead the scanner will backup automatically. The return statement in the above rule is optional. This rule might or might not produce a token.

2.4.2. How to handle include files in the parser?

The call of BeginFile can be issued by the parser as well, however, the scanner behaves exactly as in the previous case. Additionally, the lookahead tokens that the parser needs have to be taken into account. In simple cases, the parser has requested exactly one lookahead token from the scanner whenever a grammar rule is recognized or reduced. When trial parsing is used or access to arbitrary lookahead tokens via the procedures GetLookahead or GetAttribute then in general an unknown number of tokens have already been requested from the scanner.

The following example shows how to call BeginFile from the parser:

```
include_statement = ’#’ INCLUDE
                  { => { { tScanAttribute a; GetAttribute (1, & a);
                           BeginFile (a.identifier.Ident); } };
                  } identifier .
```

This works only if this rule can be recognized without trial parsing. BeginFile is called when the parser has accepted the tokens ’#’ and INCLUDE. In order to do this the parser will request one lookahead token from the scanner which is the identifier token in this case. Now it is necessary to access the attribute of the identifier token, which represents the name of the include file. The parser has not recognized this token yet, it is only known as lookahead token. The access to this attribute has to be done using the operation GetAttribute. Again the contents of the include file is inserted exactly after the last character of the identifier.

It would be wrong to move the semantic action in the above example to the end of the right-hand side. The parser would request one more lookahead token in order to recognize the complete rule. Then the include file would be inserted after this lookahead token which is usually not what is wanted.

2.4.3. How to keep track of the current filename?

Implement a stack of filenames. Assume this stack has two operations called push and pop. Whenever the procedure BeginFile is called, the operation push should be executed as well. This operation pushes a filename on the stack (see previous example). Accordingly, whenever the procedure CloseFile is called, the operation pop should be executed. However, CloseFile is usually called automatically by the scanner and therefore there is no direct way to attach statements to this call. A solution is the following: CloseFile in turn calls CloseSource. The procedure CloseSource in the module Source can be extended by an activation of pop. Get a copy of the Source module, if it does not exist already in the current directory, and modify, compile, and link it to your program. Then, the top element of the stack always contains the current filename.

2.5. How to access the start state from outside?

The current start state of the scanner is available in the variable *yyStartState*. This variable is declared static in the scanner module. In order to make it available for other compilation units you can include an interface function in the GLOBAL section such as for example:

```
int GetStartState (void) { return yyStartState; }
```

2.6. How to change the start state from outside?

The start state of the scanner is changed by statements such as

```
yyStart (Comment);
```

In C, *yyStart* is implemented by a macro and *Comment* is defined by a preprocessor constant. Both items are delcared local to the scanner module. In order to make them available for other compilation units you can include an interface function in the GLOBAL section such as for example:

```
void StartComment (void) { yyStart (Comment); }
```

2.7. How to generate efficient parsers?

First, the choice of the parser generator has some influence on the run time, although the differences are not very high. Select one of the following parser generators given in order of increasing run time:

- Use ell, version before February 1991: the error recovery detects errors as late as possible. (This tool is currently not available any more.)
- Use ell, current version: errors are detected as early as possible. This gives improved quality of error recovery at the cost of increased run time.
- Use lark.

General hints:

- Use only a few attributes for the terminal symbols, ideally at most one.
- Avoid chain rules. Example:

```
statement       = <
                = if_statement .
                = while_statement .
> .
if_statement    = IF expression THEN statement ELSE statement .

while_statement = WHILE expression DO statement .
```

  can be improved to

```
statement       = <
                = IF expression THEN statement ELSE statement .
                = WHILE expression DO statement .
> .
```

- Use ambiguous rules for expressions and specify precedence and associativity for the operators. Although this method introduces LR conflicts, run time can be decreased substantially. Example:

```
expression      = <
                = expression ’+’ term .
                = expression ’-’ term .
                = term .
> .
term            = <
                = term ’*’ factor .
                = term ’/’ factor .
                = factor .
> .
factor          = <
                = number .
                = identifier .
                = ’(’ expression ’)’ .
> .
```

  can be improved to

```
PREC            LEFT '+' '-'
                LEFT '*' '/'

expression      = <
                = expression '+' expression .
                = expression '-' expression .
                = expression '*' expression .
                = expression '/' expression .
                = number .
                = identifier .
                = '(' expression ')' .
> .
```

- Avoid separate rules for lists and list elements - merge lists and list elements: Example:

```
statement_seq   = <
                = .
                = statement_seq statement ';' .
> .
statement       = <
                = IF expression THEN statement_seq ELSE statement_seq END .
                = WHILE expression DO statement_seq END .
> .
```

can be improved to

```
statement_seq   = <
                = .
                = statement_seq IF expression THEN statement_seq ELSE statement_
                = statement_seq WHILE expression DO statement_seq END ';' .
> .
```

- Avoid separate rules for parts of constructs. Example:

```
statement       = IF expression then_part else_part END .

then_part       = THEN statement_seq .

else_part       = ELSE statement_seq .
```

can be improved to

```
statement       = IF expression THEN statement_seq ELSE statement_seq END .
```

- Avoid reduce actions. This is the general scheme behind all of the above mentioned hints on how to write grammar rules. An LR parser such as e. g. *lark* executes shift and reduce actions. It executes exactly one shift action for every input token. There is nothing that can be changed for shift actions. And it executes one reduce action for every grammar rule that is recognized. Therefore grammars should be written with as few rules as possible. All of the above examples follow this general strategy.
- For ell, use grammar rules with iteration instead of recursion.
- For lark, avoid dynamic repair of LR conflicts using semantic and syntactic predicates.
- For lark, avoid reparsing.

2.7.1. How to generate fast parsers?

- Use options -u and -o. This generates fast and large tables. The default is to generate slow and small tables.

2.7.2. How to generate small parsers?

- Use option -n. This decreases the number of case labels in switch statements.
- Use option -4. This make use of tables to decrement stack pointers instead of inline code.
- Do not use option -r: This would disable the elimination of LR(0) reductions.
- For lark, avoid right recursive grammar rules. These rules can lead to large expansion of the parsing stack.
- Preset the initial size of the parsing stack to the maximum size that will usually be needed by including a #define directive into the GLOBAL section:

```
# define yyInitStackSize        100
```

- For lark, when the default error recovery which includes comfortable reporting, repair, and recovery is not needed then it can be switched to simple panic mode by including the following definition into the GLOBAL section:

```
# define ERROR fprintf (stderr, "syntax error at %d, %d\n", \
        Attribute.Position.Line, Attribute.Position.Column);
```

- Once error recovery has been switched to panic mode, the following procedures can be removed from a generated parser: yyErrorRecovery, yyComputeContinuation, yyIsContinuation, yyComputeRestartPoints. The remaining calls to yyErrorRecovery and ErrorMessageI have to be removed, too. Also, the #include directives for the modules Errors, Sets, and Position can be removed or disabled as well, so that these modules are not linked to the program. This is achieved by including the following definition into the GLOBAL section:

```
# define NO_RECOVER
```

2.8. How to generate efficient syntax trees?

- Define the tree structure by as few nodes as possible.
- Avoid separate rules for lists and list elements - merge lists and list elements: Example:

```
statement_seq   = <
   stat_seq_0   = .
   stat_seq_1   = statement_seq statement .
> .
statement       = <
   if           = expression then: statement_seq else: statement_seq .
   while        = expression statement_seq .
> .
```

can be improved to

```
statement_seq   = <
   no_statement = .
   statement    = next: statement_seq <
      if        = expression then: statement_seq else: statement_seq .
      while     = expression statement_seq .
   > .
> .
```

- Use inline attributes. Example:

```
call            = identifier arguments .
identifier      = [ident: tIdent] .
```

  can be improved to

```
call            = [ident: tIdent] arguments .
```

- Generate only the procedures that are really used.
- Switch off the generation of the procedures CheckTree or WriteTree after the development phase.
- Remove the array Tree_NodeName from the generated tree module in case it is not needed. (This should be supported somehow!)
- Remove #include directives for unused modules of the library reuse from a generated tree module. By default, almost every module from the library reuse that defines a data structure is included which may lead to the linking of superfluous modules. (This is done automatically.)

2.9. How to release space for parts of syntax trees?

In order to be able to free space for single nodes or subtrees using the procedure ReleaseTree two macro definitions are necessary in the GLOBAL section of an ast description. For example:

```
# define yyALLOC(size1, size2) (tTree) Alloc (size2)
# define yyFREE(ptr, size) Free (size, ptr);
```

The section 2.15. of the *ast* manual about "Storage Management" describes the details.

2.10. How to generate efficient attribute evaluators?

- Use attribute grammars from the class OAG. This means to avoid the class WAG and the option -L.
- Use the option -0 which optimizes the storage for attributes.

3. How to avoid memory leaks?

Almost all modules of the library reuse that store data have functions for initialization and for finalization. The function names uniformly start with Begin or Close:

Module	Initialization	Finalization
Idents	BeginIdents	CloseIdents
StringM	BeginStringMemory	CloseStringMemory

rMemory	BeginrMemory	CloserMemory
Errors	BeginErrors	
Sets	BeginSets	

In simple cases these functions do not have to be called because the modules are initialized automatically, by default. However, if a parser is to be executed repeatedly within a loop then explicit initialization and finalization is advantageous in order to avoid memory leaks. The order of the calls for initialization and finalization is significant and should be as follows:

```
BeginrMemory        ();
for (;;) {
   BeginStringMemory ();
   BeginIdents       ();
   BeginErrors       ();
   Parser            ();
   ReleaseTreeModule ();
   CloseIdents       ();
   CloseStringMemory ();
}
CloserMemory        ();
```

4. How to redefine the exception handling?

Some of the Cocktail generated source modules might detect serious error conditions which make it impossible to continue program execution. These conditions are regarded as exceptions. The default reaction is to emit an error message and to terminate program execution. The default reaction can be changed by the user as described below. The following kinds of exceptions might be detected:

Module	Exception	Remark
Scanner	cannot open input file	should be checked before calling BeginFile
	out of memory	huge token or lookahead, too many nested include files
	too many calls to CloseFile	can be checked at programming time
	too many calls to yyPop	can be checked at programming time
	internal error	should not occur
Tree	out of memory	too many tree nodes
Trafo	function failed	can be avoided at programming time
Errors	fatal message	can be avoided at programming time

The following example code shows how to change the default reaction of the exception handling using the routines *setjmp* and *longjmp* of the C library:

```
# include <setjmp.h>

jmp_buf jmp_env;        /* buffer for longjmp */

void OurExit (void) { longjmp (jmp_env, 1); }   /* goto exception handler */

int main () {
                        /* register exception handler */
   if (setjmp (jmp_env)) {
      fprintf (stderr, "caught exception0); return 1;
   }
                        /* redefine ..._Exit routines of all Cocktail modules */
   Errors_Exit          = OurExit;
   Scanner_Exit         = OurExit;
   Tree_Exit            = OurExit;
   Trafo_Exit           = OurExit;

   ...
}
```

The above few lines at the beginning of the procedure *main* redefine the exception handling to execute the "then-part" of the if statement in case of an exception. It can contain arbitrary code.

5. How to compile the Cocktail code?

5.1. How to compile the Cocktail source code?

In general every good 32 bit C compiler should work. The data type ’int’ should be represented by 32 bits. The compiler should obey either the C language definition of Kernighan and Ritchie or ANSI C. And it should have no restrictions such as for example array size <= 64 K or stack size <= 64 K. Under Unix, the command ’make’ at the global level of the source distribution should do the job (see file README). (Compiling the Cocktail source code with the 16 bit compiler Microsoft Visual C++ version 1.52 failed.)

5.2. How to compile the source code generated by Cocktail tools?

See answer to previous and next questions.

5.3. How to compile the generated code with Microsoft Visual C++?

Since version 9607 it is possible to compile the source code generated by Cocktail with the Microsoft Visual C++ compiler version 1.52. In order to overcome the restrictions of this 16 bit compiler the following hints might help:

- select the huge memory model (option /AH)
- set the new segment data size threshold (option /Gt8192)
- increase the stack size (up to 64 K)
- implement attribute evaluators as stack machines (ag -K)

Contents