A Common Lisp CGOL

Tom Phelps

December 16, 1992

Abstract

CGOL is an Algol-like notation for Lisp. The original version, written by Vaughan Pratt at M.I.T. in the early 70s was written in Maclisp. This new version is based on Common Lisp. CGOL was translated to Common Lisp in the following four-stage process:

(1) `cgol.tok`, the tokenizer has been almost completely rewritten; (2) `cgoll.l`, the main translation loop with library of translation schemas has been converted from Maclisp to Common Lisp; (3) the code that `cgoll.l` *produces* has been converted to Common Lisp; (4) selected examples of CGOL programs themselves were rewritten, since certain aspects of the semantics of Maclispwould otherwise not be modelled in the expected fashion. Maclisp differs from Common Lisp in a variety of respects, and some of them are apparent from CGOL including direct escapes to Lisp, variable scoping, function definitions and numerous other aspects.

In contrast to the *programming* described above, the major contribution of this *paper* is annotation of selected code from the CGOL translator.

Contents

1 Introduction

CGOL is a notation for writing Lisp in Algol-like notation. This paper describes the Common Lisp translation of this system in great detail and gives a brief users guide in its use. Most (but not all) of the original CGOL language remains valid; before trying to use the Common Lisp version, one should read the original manual (`cgol.man`). The paper "Top Down Operator Precedence" [Pra73] describes the theory behind (much of[1]) of the parsing.

2 Limitations of CGOL

As described in [Pra73] and explained in great detail in the annotated code, CGOL uses a easy-to-implement method of parsing. In contrast to LALR parsers which use complex theory and analysis to produce a efficient, table-based parser, the heart of CGOL's parsing technology can be written in a handful of lines.

```
(defun parse (rbp)
  (do ((translation (funcall (nud)) (funcall (led) translation)))
      ((not (< rbp (or (getden lbpl) 0))) translation)
   nil))
```

Just as in Lisp, attached to every functional symbol is a property holding its implementation. Except for the first token in an expression, which is called without it, `parse` calls the functional translation of the current operator along with the left-hand side of the expression built up thus far. For instance, when the parser reaches the plus sign in `a*b*c+4` it has accumulated the running translation of `a*b*c` and passes this to `+`, which parses its right-hand sign to obtain its right-hand operator.[2]

The obvious questions is, If CGOL is so simple and yet powerful enough to (correctly) parse complicated expressions with the correct operator precedence, why isn't CGOL used more widely?

2.1 Full Correspondence

After all, CGOL is simply another *notation* for Lisp. In fact, the CGOL manual defines the meaning of CGOL expressions in terms of their Lisp translations! CGOL's value is that its notation is, to some, more aesthetically pleasing than Lisp's—with all those unsightly parentheses! The problem is that, except for simple cases, to use CGOL one must more or less know the meaning of the Lisp translation, which begs the question, Why not dial direct? Say, for instance, there is a complex Lisp operator to which CGOL provides a more aesthetically pleasing interface—arrays, for instance.

[1]Parsing of complex forms is done on an ad hoc basis.
[2]More details are given with the annotated code.

```
make-array dims [:adjustable adj] [:element-type et] [:fill-pointer fp]
  {[:initial-element ie] | [:initial-contents ic] | [:displaced-to dt
  [:displaced-index-offset dio]]}
```

To use the CGOL notation I must first figure out how I wish to utilize this function (as I have forgotten one of the seven optional parameters) so I look up this function in Lisp (which is how CGOL defines its notation). Once having determined the Lisp form I need, I must translate this into CGOL. Why not cut out the middle man? To be sure, it would ease the life of the CGOL implementor as optional parameters are handled in an ad hoc way, with special-purpose code included with every function.

A more egregious example of a Lisp function to which CGOL could add little value is `loop`. Already this function looks more like its iteration counterparts in imperative languages than anything else found in Lisp. If the primary value of CGOL is to provide an imperative-like interface to Lisp, what's left for it to do? (The original CGOL predated this elaborate `loop` construction of Maclisp and then Common Lisp.)

Another rationale for a top-level non-Lisp language that is translated into Lisp is that the top-level language might disguise differences in the underlying dialect. The RLISP and PSL ("Portable Standard Lisp") dialects were used for such a purpose in making a portable substrate for the REDUCE computer algebra system (ref: Martin Griss) Similar arguments could be made for the GLISP work by (ref: Gordon Novak).

In such a system one would localize the CGOL translator for one's particular implementation of Lisp and then be able to translate arbitrary other CGOL programs with no further grunt work needed. Sadly, this could not have been an important rationale for CGOL since it was not widely ported to dialects other than Maclisp and the very-similar Franz Lisp, and CGOL programs tend to leak too directly into the underlying semantics to cover much.

2.2 Efficiency

CGOL is faced with a design decision to either provide a one-to-one correspondence with Lisp or abstract a range of similar Lisp functions (for example, `if` and `cond`, or iterators) and try to translate back into the most efficient form. With iterators, CGOL produces lousy code. It simply fills in the relevant parts of a `do`, when it could use `dotimes` or `dolist`. At other times, it does much better. For instance, nested `if`'s are collected together into a single `cond`. Regardless of specific examples where CGOL is able to achive an efficient form,[3] the observation is that this always takes custom, special-purpose, error-prone code to support. And if the user knows the more efficient form at the outset, he

[3] Of course, who knows what `dolist` and `dotimes` expand into—probably `do`! The point is that the more information the compiler receives, the better chance it has of optimizing the code.

can't specify this (without sidestepping CGOL althogether by using its escape into Lisp).

2.3 Comparison with LALR

Since CGOL's (top-down) operator precedence algorithm is trivial to implement and so lightweight, why are most compilers today written with a LALR parser generator (such as YACC or Bison)?

For one, LALR is more powerful. As the CGOL manual states:

> For those accustomed to BNF, the grammar of CGOL might look like:
>
> ```
> <expression> ::= if <expression> then <expression> [else <expression>]
> <expression> ::= <expression>=<expression>
> <expression> ::= (<expression>)
> <expression> ::= <expression>;<expression>
> ```

Everything is reduced to an expression! This is not helpful in constructing parse trees (actually AST trees) for later stages of the compiler.

An LALR parser can correctly distinguish identical prefixes of a parse string, whereas CGOL must make its decision upon first sighting a particular token in the token stream.

For another reason, LALR accomodates a more regular specification of legal parse strings. CGOL widely separates its definitions in sometimes complicated list building functions. There is no crossreferencing of these functions, so a function can be multiply defined. With a LALR grammar generator, much error checking can be done at grammar compile time, in locating multiply-defined functions, and finding shift-reduce and reduce-reduce errors. Since CGOL must make its decision at each token, it "isn't sophisticated enough to have shift-reduce and reduce-reduce errors", as the most recently defined function for that token is the one called.

3 Selected Annotated Common Lisp CGOL Code

CGOL takes full advantage of Lisp's fluidity between code and data to manipulate the token stream to obtain a valid Lisp expression, which it hands off to Lisp to do the evaluation. Thus, CGOL is basically less than the front end of a compiler, as it does the lexical analysis and a simple parsing The parsing results in an "intermediate form" that is a Lisp version of the program. The Lisp system then evaluates that program.

The code below intermixes program code with explanation thereof, in the style of Norvig [Nor92].

This section only presents some of the most interesting or complicated code. Of course, the entirety of CGOL is commented in the source files.

3.1 Preliminaries

3.1.1 r-e-p loop

Below is the main read-eval-print loop.

```
(defun toplevel-parse (standard-input
  (format t "~%cgol(1)> ")
  (let ((expr 'do-at-least-once) (show-syntax t))
    (setq cgolerr nil)
    (setq parser-debug nil)
    (setq scripting nil)
    (do ((ctr 2 (+ ctr 1)) (bozo (advance) (advance))) ((not expr) 'ok)
      (setq expr (parse -1))
      (cond ((eq expr 'eval) (setq syntax-needed (not syntax-needed)))
    ((eq expr 'show) (setq show-syntax (not show-syntax)))
    ((eq expr 'parser_debug) (setq parser-debug (not parser-debug)))
    ((eq expr 'scripting) (setq scripting (not scripting)))
    ((eq expr 'quit) (setq expr nil))
    (t (if show-syntax (format t "~%  Lisp> ~a " expr))
       (if syntax-needed (format t "~%  Value> ~a" (eval expr)))))
      (format t "~%cgol(~d)> " ctr)
      )))
```

The user invokes this function (possibly through its abbreviation, `(cg)`) with evaluation mode on, show Lisp translation mode on, and the two parsing modes, which show the individual tokens as they are read in, off. The `do` accepts an expression, parses it, and then examines the output to decide if it is an internal command or it should evaluate it.

3.1.2 Convenience Functions

Maclisp has a more direct way of attaching properties, especially the function property of a symbol. Common Lisp has a more verbose `setf-get` construction, but I defined macros to achieve the same conciseness of expression. These macros could not be replaced by function calls here, because we do not want the function parameters to be evaluated.

```
(defmacro defnud (fn-name fn-lambda)
  `(setf (get ,fn-name 'nud) ,fn-lambda))
(defmacro defled (fn-name fn-lambda)
  `(setf (get ,fn-name 'led) ,fn-lambda))
```

3.2 Lexical Analysis

The lexical analyzer is that part of a compiler[4] that accepts the program input (usually a stream of characters) and produces tokens for a later stage.

[4] And CGOL is a compiler, not from a high level language to machine, but from one high level language to another, Lisp.

I almost completely rewrote CGOL's tokenizer (`cgtok.l`), letting the Common Lisp tokenizer handle all that it could, including floating point numbers. `cgtok.l` was a hand-written tokenizer which modified the read table so that every character was redirected to it. I tried to keep the Common Lisp read table as much as possible, only accepting CGOL-specific constructions.

The core of the tokenizer is comprised of the following functions.

```
(defvar *cgol-readtable* (copy-readtable))

(let ((*readtable* *cgol-readtable*))
  ; unchanged are: string, numbers, whitespace
  (set-macro-character #\% #'read-comment nil)
  (set-macro-character #\! #'(lambda (s c) (lisp-read s)) nil)
  (set-syntax-from-char #\? #\\) ; escape character is '?'
  ; semicolon is now statement terminator--
  ; changed by initialize-multi-character-token-table
  )

(defun initialize-multi-character-token-table (string)
  (setq ctoken-table string)
  (every #'(lambda (c)
     (set-macro-character c #'smash-token nil *cgol-readtable*))
     string))

(defun cread (&optional (stream *standard-input*))
  (let ((*readtable* *cgol-readtable*))
    (read stream)))
```

Here, we make a new read table by copying the standard Lisp read table. Some tokens are redefined right away: those for comments (%), escapes into lisp (!), and character escapes (?—same as Lisp's \). The lisp function `initialize-multi-character-token-table` is called from user level to define certain characters as operators. These characters will then dispatch to the function `smash-token` which deals with collecting all the pieces of multicharacter-tokens. Through great hackery lamented elsewhere, it correctly parses the $\geq$ in `1>=3`, and `1>+3`. With this support structure, the call to obtain the next CGOL token (`cgoltoken`) reduces to simply a call to `cread`!

I deleted the `cons` cell conservation functions, because `cons` appeared not used wildly (certainly not by the built-in functions); the code generates only a small amount of garbage, and modern generational garbage collectors are alleged to use only takes a small percentage of system time anyway.

The dreadful means of accumulating multiple character tokens is shown below.

```
(defun smash-token (stream c)
  (intern (coerce
   (cfollow-tail c stream
            (subseq ctoken-table
              (1+ (position c ctoken-table))))
   'string)))
```

```
(defun cfollow-tail (c stream table)
  ;; this way of recognizing tokens is taken from the original cgol,
  ;; is fast and easy and passes all tokens which are subtokens
  ;; of explicitly defined tokens.
  ;; [but it will pass erroneous multi-character tokens.  --tap]
  (let* ((c2 (char-upcase (peek-char nil stream)))
 (posn (position c2 table)))
    (cons c
  (if posn
      (progn (read-char stream)
     (cfollow-tail c2 stream
                     (subseq table (1+ posn))))))))

(initialize-multi-character-token-table "$|-+#&;)(*,'/:<=>@[\\]^`{|}~")
```

Upon receiving a token from the reader, `smash-token` searches for a suitable followup in the stream supplied by `initialize`. *We can match a character with any token that follows.* Not only ugly, it can return erronous tokens! What would be better is a list of tokens, each self contained. However, it would still take a bit of character pasting to determine that `1<+2` should yield four tokens `1 < + 2`, while `1<=2` should return a compound $\leq$ for three tokens `1 <= 2`. In his baby Macsyma implementation, Norvig [Nor92] circumvents this problem by insisting that tokens be separated by white space. A finite state machine solution, such as that supplied by `lex`, would be useful here.

3.3 Operator Precedence

The following functions form the heart of CGOL's operator precedence structure. `parse` was considered earlier.

```
(defun advance nil
  (setq stringnud nil)
  (setq token (cgoltoken))
  (if (stringp token)
      (setq stringnud #'ret-tok token (intern (string-upcase token))))
  (if parser_debug (format t "=~a= " token)) ; let's see the tokens
  (if scripting (format t "~a " token))
  token)


(defun nud nil
  (or (verify (or stringnud
  (cond ((numberp token) (list 'lambda nil token)) (t (getden nudl)))))
      (nuderr)))


(defun getden (indl)
   (and indl
(or (and (symbolp token) (get token indl))
    (getden indl))))
```

`advance` is called by parse to obtain the next token. It also establishes certain flags (and handles the parsing debugging output). Strings can have special significance (functions identify the function name and other delimiters this way), and so they set a special flag; the interned symbol version of the string is then stored in `token`. For uniformity with operators, which are evaluated by `parse`, everything is returned to `parse` is a function. The case of strings and numbers amounts to a closure which returns that value when applied. If the token is a symbol, `genden` is called to obtain its functional definition. As a single symbol may have a different function for both nud and led (i.e., for whether it takes a preceding expression as an argument or not; for example, logical not is a nud, but arithmetic plus is a led), the type of function desired is passed to `getden`.

3.4 Translation Library

3.4.1 define

Because of its great size `define` has been given its own page, which may be detached to aid in following along with the following below. `define` is so large because it handles a great variety of function definitions. The basic form of a `define` is this:

```
define <pattern> [,bp [,bp]]; body
```

Some examples are:

```
define "F"(x,y), 14; x**2 + y**2 $
define "LOG" a "BASE" b; ...$
```

where `F` defines a function with a left binding power of 14 that takes two argument and returns the sum of their squares. The second function defined is `log` which takes two arguments. `F` is called by F(3+2, a), for example. `log` also must be called in exactly the form it was defined but it now contains a delimiter between the two arguments, thus `log 8 base 2` is legal whereas `log(8,2)` is not.

Before considering the code, note that an operator is defined by a triple: (1) its syntactic form, (2) its function (body), and (3) its rank in the precedence hierarchy.

Now considering the code, found on the next page, the first `cond` determines whether the function is a nud or a led: if is a nud if the first token is the function name (a string) or it is a single character followed immediately by its parameters (indicated by left parenthesis); otherwise it is a led.

Before coming to the parameter list, we may have delimiter-separated tokens; delimiters are checked for and formal parameters are collected in the list `argts`. Next we gobble up the parameter list, a comma-separated list of token between parentheses.

Binding powers are optional, and nud's don't even have left binding powers. Next the function (which will be constructed later) is attached to the main CGOL parse engine by defining its nud or led property, whichever is appropriate, to (when invoked) parse the token stream to obtain the arguments (and check for the delimiters) and then to call the actual function.

A body may or may not be supplied, as this may be an actual function definition or only a declaration for one of a mutually recursive pair. If there is a body, it is parsed recursively, using the other definitions in the translation library, and it is defined in a `defun` form.

`define` is a case where the implementation is so complex that the power of LALR is sorely missed.

3.4.2 loop

Basically, Lisp's `loop` is already Algol-like; it certaintly isn't Lisp-like. Rather than come up with a stylized version, we should just provide a transparent interface. But this already exists!

4 Implementation Notes, Usage and Examples

4.1 Implementation Notes

As the paper demo (in appendix A) amply demonstrates, CGOL now lives in Common Lisp. However, due to this evolution and natural selection some pieces of the original implementation have not survived. In particular, `array`, `let`, and the I/O functions have changed enormously from Maclisp to Common Lisp. The old use of arrays was not part of the translation proper, but rather a reference to CGOL's version of arrays. To translate arrays properly, one would probably want to support all of the optional parameters and features of Common Lisp. This would involved complicated code to support it and associated documentation to describe the relation between the CGOL and Common Lisp versions. But, as mentioned under limitations above, if we are going to mimic all of the features of Common Lisp, why not dial direct by using the escape into Lisp?

In the old style, `new` declared the local variables. Under Maclisp, these were special variables and the CGOL programs used this fact. With Common Lisp, however, these variables are lexically scoped, whereas `let` is connected to a weird lambda form. Depending on one's tastes, `let` could be remapped to the familiar `let` of Common Lisp.

Finally, the I/O have been completely redone in Common Lisp. Maclisp used `UREAD`, `UWRITE`, `FILE` and others.

If one were to enhance this version in the direction of robustness (say, in preparation for wide distribution) we suggest the following the CGOL changes.

- The method of operator lexical analysis of the original implementation was maintained in the Common Lisp version.[5] This begs for a `lex`-like, finite-state-machine-based analyzer. (Refer back to the annotated code for a full lament of this code.) (ed. note: Couldn't this be done in a Lisp-like manner through read-table hackery? RJF)
- As mentioned in limitations above, CGOL code itself must be rewritten. Perhaps the most interesting example of CGOL code is `CGOLPRINT`, which translates *from* Lisp *to* CGOL. This entails massaging the CGOL code to fit comfortably in Common Lisp, and a more thorough reworking to have that code recognize and translate Common Lisp instead of Maclisp. The most extensive reworking that an average CGOL file needs is a re-orientation from special variables to lexical scoping, with the necessary parameters passed explicitly, rather than through dynamic scope. The other outstanding feature of CGOL code is a metacircular interpreter for CGOL itself. One must already have a CGOL implementation for the metacircular to work (it does not bootstrap from scratch), but given the seemingly mandatory nature of metacircular interpreters for Scheme or subsets of Lisp in *any* Lisp text with a practical dimension, a CGOL metacircular interpreter appears *de riguer*.

4.2 Usage

Obtain copies of the Common Lisp files files `parser.cl` and `cgol.cl`. Load them into Lisp in that order. Then type `(cg)`. Upon initial entry, the translator is set to both (1) show the Lisp translation of the CGOL expression and (2) evaluate it. This is usually what one wants, but viewing the translated expression can be toggled on and off by typing `show $`. If one is experimenting with CGOL and does not wish to worry about variables or functions being undefined, type `eval $` to toggle evaluation off or on. When preparing a paper demo, this author found that `script` only reported text printed to the screen, not text read from the file; he created the option `scripting $` to echo the tokens back to the screen as they are read from a file. `parser_debug $` does much the same, except the tokens are made to stand out to the eye by enclosing each in a pair of equals signs.

Appendix A contains a paper demo of the CGOL translator in action. Remember that every command must end with a dollar sign. `quit $$` exits back to Lisp.

4.3 Examples

Refer to the paper demo for more elementary examples. Here we first consider a led ("left denotation") taken from the manual, along with its translation into

[5] In fact, this is basically the only part of the parser code which wasn't completely rewritten.

(Common) Lisp.[6]

```
define a "bozo" b, 14, 13;
if a then car a ^ (cdr a @ b) else b $

(PROGN
 (DEFLED 'BOZO #'(LAMBDA (LEFT) ('BOZO (PROG1 LEFT) (PROG1 (PARSE '13)))))
 (DEFLBP 'BOZO 14)
 (DEFUN BOZO (A B) (COND (A (CONS (CAR A) (APPEND (CDR A) B))) (T B))))
```

229z First notice that the translator locates the function name by searching for a string; non-strings are regarded as parameters, hence the function definition of `bozo`, with parameters `a` and `b`. The body of the function is translated into its Lisp equivalent and forms the body of the Lisp function. The numbers 14 and 13 define the left binding power and right binding power, respectively. The left binding power is saved as a property by the `deflbp` line. The right binding power is called into play when the function is seen by the CGOL translator. When the translator encounters the symbol `bozo` it extracts its `led` property and calls the function with the left argument; the function then calls parse with its right binding power to obtain its left argument.

```
define "log" a "base" b ; a+b$

(PROGN
  (DEFNUD 'LOG #'(LAMBDA NIL
    ((PROG1 'LOG) (PROG1 (PARSE '25)) (CHECK 'BASE)
     (PROG1 (PARSE '25))))) NIL
  (DEFLBP 'BASE 0) (DEFUN LOG (A B) (+ A B)))
```

Here we have simplified the body of the function and omitted binding powers (which are given default values), in order to concentrate on the pattern matching. As above, the (first) string determines the function name and one can see from the Lisp translation that the first argument is accumulated as before by a call to `parse`. Now, however, we must match the pattern extended by `base` and then parse the second and final argument.

The following example is drawn from the paper demo, which in turn was taken from [ASS85].

```
define "FIB" (N) ;
if (n=0) then 0 else if (n=1) then 1 else fib(n-1)+fib(n-2) $

(DEFUN FIB (N)
  (COND ((= N 0) 0)
        ((= N 1) 1)
        (T (+ (FIB (- N 1)) (FIB (- N 2))))))

define "fib_i" (n) ;
fib_i_aux(1,0,n) $
```

[6] The output has been slightly reformatted from that produced by the translator.

```
define "fib_i_aux" (a,b,cnt) ;
if (cnt=0) then b
else fib_i_aux(a+b,a,cnt-1) $

cgol(29)> FIB ( 20 ) $
  Lisp> (FIB 20)
  Value> 6765

cgol(40)> fib_i(20)$
  Value> 6765
```

Here we have defined two versions of a function which determines the n^{th} Fibonacci number, the first recursively and the second iteratively (using recursion for its looping construct). The recursive (first) version demonstrates how `if`'s nest nicely: instead of multiple `if-else` pairs, CGOL constructs a single `cond`. The iterative version shows how to declare and use multiple parameters. Note that a function is `define`d with its name as a string (i.e., with quotes) but it is used as a symbol (without quotes). Of course, both functions return the same answer for the same parameters, although the doubly-recursive Fibonacci function is the only one to take noticeable calculation time (on a Sparc 1+) of any function in the test suite.

And of course functions compose.

```
cgol(9)> fib_i(7) $
  Lisp> (FIB_I 7)
  Value> 13

cgol(10)> fib_i(13) $
  Lisp> (FIB_I 13)
  Value> 233

cgol(11)> fib_i(fib_i(7))$
  Lisp> (FIB_I (FIB_I 7))
  Value> 233
```

Again, refer to the paper demo for a wider range of examples.

A Demo Script

References

[ASS85] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 1985.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[CRM80] Eugene Charniak, Christopher K. Riesbeck, and Drew V. McDermott. *Artificial Intelligence Programming*. Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersey, 1980. Provided information on `lexpr`'s and `fexpr`'s.

[Fra85] Franz, Inc. *Franz Lisp Reference Manual*, opus 42 edition, 1985.

[Nor92] Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann Publishers, San Mateo, California, 1992.

[Pra73] Vaughan R. Pratt. Top down operator precedence. In *ACM Symposium on Principles of Programming Languages*. Massachusetts Institute of Technology, October 1973.

[Pra77] Vaughan R. Pratt. Cgol - an algebraic notation for maclisp users. online documentation file, January 1977.

[Ste90] Guy L. Steele Jr., editor. *Common Lisp: the Language*. Digital Press, Bedford, Masachusetts, second edition, 1990.