Is Forth Code Compact? A Case Study

M. Anton Ertl
Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8, A-1040 Wien
anton@mips.complang.tuwien.ac.at
http://www.complang.tuwien.ac.at/anton/
Tel.: (+43-1) 58801 18515
Fax.: (+43-1) 58801 18598

Abstract

Forth advocates often claim that Forth code is smaller, faster, and requires less development time than equivalent programs in other languages. This paper investigates this claim by comparing a number of parser generators written in various languages with respect to source code size. The smallest parser generator (14 lines) in this comparison is written in Forth, and the other Forth program is smaller than the others in its class by a factor of 8 or more; however, the Forth programs do not have all the features of their counterparts. I took a closer look at Gray (in Forth) and Coco/R (in Modula-2) and found that several Forth features missing from Modula-2 give Gray more than a factor of three advantage over Coco/R (even if the other size differences were solely due to differences in functionality): run-time code generation; access to the parser and a simple, flexible syntax; and Forth's dictionary.

1 Introduction

Forth advocates claim that Forth programs are smaller, faster, and are developed faster than programs in other languages, by a wide margin. If they have these properties, they are not due to the language implementations [EM95], but due to the way these programs are designed.

In contrast, software engineering people believe that the programming language does not matter for the design and consequently the speed of development (coding supposedly takes only 20% of the time). And Forth programs would not have any size and speed advantages over other languages, because the design would be the same as for other languages.

So we would like to test whether the programming language, in particular Forth, does play a role in the design, and consequently in code size, speed and development time. One way to check this would be to have two teams solve the same problem in two different programming languages, and then compare the development time and the solutions; the teams should be equally competent in their respective programming languages.

Unfortunately, equally competent teams are hard to find; actually, it's even hard to determine whether two teams are equally competent. To solve this problem, one could use a number of teams for both languages, allowing to use statistical means to separate the differences due to the language from the differences due to variations in competence; however, given the very high variation in competence between teams, we would need a big sample to make the confidence interval small.

If the problem to be solved is of significant size, a study that used just two teams would cost a lot of money; with a big sample, the cost would be astronomical.

As an alternative, in this paper I compare the solutions written in different languages (including Forth) for a specific problem: parser generators. The advantages of using this application for such a case study are:

- It has been implemented many times and thus provides a big sample for doing a comparison; e.g., there are at least three solutions in Forth for this problem [Rod90, IH90, Ert97a], even though usage of a parser generator is rather unidiomatic for Forth.
- The problem is somewhat sophisticated, eliminating generally incompetent implementors (although this tells us nothing about the competence in a specific language).

The disadvantages of this approach are:

- Each developer tried to solve a somewhat different problem when building the parser generator, so they are not entirely comparable. However, some of these issues also occur in the real world with real customers, as different groups

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in: EuroForth '99 Conference Proceedings

will get different results out of requirements engineering; it may even be that the parser generator problem, being well-understood, has less variation in the requirements than other problems.

- I can only compare the resulting products (e.g., size, speed), not their development (e.g., development cost).

In this paper I compare a number of parser generators superficially (Section 2), and two more deeply (Section 3).

2 The Parser Generators

The parser generators I compare are listed in Fig. 1, together with the URLs where you can find them.

The selection criteria were: The source code had to be freely available, and I had to find it in my Web search (I started with http://www.idiom.com/free-compilers/, but also used Altavista). I concentrated on those programs that generate recursive-descent parsers, because I intended to compare them with Gray (a generator of recursive-descent parsers in Forth [Ert97a]). I selected Bison because of its (and yacc's) popularity. I selected bnfparse, because it is written in Forth, and DCG to have something to compare it to. Mop, kwParsing, and mlyacc were selected because of the languages they are written in, to get a wider spectrum of languages.

Figure 2 shows the language(s) and size of the parser generators. The *implementation language* is the language in which the parser generator is written; Most parser generators produce *output* as source code, but Gray and kwParsing produce it as binary code or binary tables, and bnfparse and mop do not generate code, but execute the grammar directly.

Source lines of code (SLOC) are used as a metric of code complexity; source lines are the lines in the source code, excluding blank lines and comments[1].

While this metric has some shortcomings (in particular, the error introduced by the differences in programming style), it is relatively easy to measure, widely used, easy to understand, and important in the real world: programmer productivity is unfortunately often measured by the lines of code produced, and I have read claims that the number of bugs is proportional to the number of lines of code, even across languages. Since this metric is the central issue in this paper, the parser generators are sorted by this metric throughout the paper, making it easier to see the correlation between size and other factors.

[1] I only filtered out regular comments, not text commented out by using conditional compilation.

Figure 2 shows three orders of magnitude of size difference. What are the reasons for this enormous difference? The first hypothesis is that the programs differ in size because they do something different, i.e., that there are differences in functionality. So, in Fig. 3 we look at what the programs do:

algorithm The algorithm determines what kind of grammars can be used, how many problems (e.g., ambiguous grammars) are reported at parser generation time, and how fast the resulting parsers are.

The most restrictive grammar class in our selection is LL(1). Other grammar classes can handle more grammars in theory, but often are not better for languages occuring in practice; one exception is the pred-LL(k) class of ANTLR, whose syntactic and semantic predicates are modeled on the hacks that are used for dealing with problematic languages. The power of backtracking top-down parsers is between LL(1) and LL(k) with syntactic predicates, depending on how much inefficiency in the parser you find acceptable. SLR(1) is better than LL(1) and LALR(1) is better than SLR(1).

Apart from backtracking parsers, all parser generators report if a grammar does not belong to the grammar class it can handle, and usually provide ways to resolve (or ignore) the conflict. For backtracking parsers, such problems are only noticed during parsing: they can result in exponential run-time or in endless loops.

Recursive-descent parsers are usually fastest, followed by table-driven parsers, and then by backtracking parsers.

EBNF Some parser generators provide only simple BNF (i.e., terminals, nonterminals, sequence, and alternative), while others provide additional constructors (option, and various repetitions); EBNF does not increase the number of languages that can be expressed, but it makes their expression easier and shorter.

data flow Plain parsers just accept or reject a string. We usually want to do more processing, e.g., interpret the program represented by the string, or generate code for it. To do this, the parser specification usually specifies actions that are executed when the corresponding rule is used in parsing[2].

It is necessary to pass data between the actions (e.g., the result of evaluating a subexpression).

[2] Attribute grammars are a more general approach, but the parser generators examined here all just have actions, sometimes disguised with attribute grammar terminology.

Program	Version	Paper	URL
bnfparse		[Rod90]	http://www.zetetics.com/bj/papers/bnfparse.htm
DCG			http://www.complang.tuwien.ac.at/clp/dcg.pl
mop	1.06		http://www.oasis.leo.org/perl/exts/devel/mop.dsc.html
Gray	4	[Ert97a]	http://www.complang.tuwien.ac.at/forth/gray4.zip
kwParsing	1.0		http://www.chordate.com/kwParsing/
Coco/R	1.39	[Han90]	ftp://ftp.inf.ethz.ch/pub/software/Coco/
mlyacc	110		ftp://ftp.research.bell-labs.com/dist/smlnj/release/
rdp	1.5		ftp://ftp.dcs.rhbnc.ac.uk/pub/rdp/
bison	1.27		ftp://ftp.gnu.org/pub/gnu/bison/
ell	9208		ftp://ftp.gmd.de/gmd/cocktail/
ANTLR	1.33		http://www.polhode.com/pccts.html

Figure 1: Compared parser generators

Program	Implementation Languages	Output	Source Lines	All Lines	Remark
bnfparse	Forth	-	14	16	
DCG	Prolog	Prolog	68	226	
mop	Perl	-	156	291	only Rule.pm (not Lex.pm)
Gray	Forth	bin. Forth	473	754	
kwParsing	Python	bin. Python	1691	2883	
Coco/R	Modula-2, Coco/R	Modula-2	4005	5106	
mlyacc	ML, mlyacc	ML	4395	6352	
rdp	C, rdp	C	4947	6519	
bison	C	C	7806	11258	
ell	Modula-2, Cocktail tools	Modula-2, C	8712	11384	
ANTLR	C, ANTLR	C	18577	23318	

Figure 2: Programming languages and sizes

Program	algorithm	EBNF	data flow	scanner	error recovery	other
bnfparse	top-down backtracking	-	-	-	-	
DCG	top-down backtracking	-	L+	-	-	logic variables
mop	top-down backtracking	$\surd$	L	-	-	
Gray	LL(1) recursive descent	$\surd$	L	-	-	
kwParsing	SLR(1) table-driven	-	S	$\surd$	-	
Coco/R	LL(1) recursive descent	$\surd$	L	$\surd$	$\surd$	
mlyacc	LALR(1) table-driven	-	S+	-	$\surd$	first-class functions
rdp	LL(1) recursive descent	$\surd$	L+	$\surd$	$\surd$	tree-building support
bison	LALR(1) table-driven	-	S(L)	-	$\surd$	
ell	LL(1) recursive descent	$\surd$	L	-	$\surd$	
ANTLR	pred-LL(k) recursive descent	$\surd$	L	-	$\surd$	tree-building support

Figure 3: Functionality of various parser generators

Most parser generators support passing data from left to right, both up and down the parse tree (known as L-attribution). However, this is not normally possible with bottom-up parsers (SLR, LALR); they usually support only passing data upwards (towards the root) in the parse tree (known as S-attribution); there is a way to pass data downwards in bison/yacc, but it is hard to use and it can cause conflicts in the grammar (you are only safe from that if the grammar is LL(1), so you lose the LALR(1) advantage). In mlyacc this trick is not possible; instead you can pass a function upwards; you emulate passing a value downwards by passing it to this function.

You can have actions in bnfparse, but there is no provision for passing data around (other than using variables). The multiple passes possible in rdp and the logic variables used in DCG provide somewhat more powerful data flow capabilities than plain L-attribution.

Gray and ANTLR allow passing *multiple* values upwards, whereas the other parser generators are restricted to passing one value upwards, requiring to box the values in structures (and this may not be easily possible in the Modula-2 generators).

scanner Several parser generators include a scanner generator, whereas others rely on separate tools and allow hand-written scanners. Mop comes with a scanner, but it was easy to separate it, so I counted only the lines for the parser. In the other cases it was not so simple to separate the scanner generator from the parser generator, so I chose to count the whole thing.

error recovery When the parser encounters an error, it would be nice if it could continue to find more errors (or at least it was nice, when each new run took a noticable amount of time). Several parser generators support this with error recovery schemes, each one different, requiring more or less programmer support. The most sophisticated scheme appears to be ell's, but I do not know if it has an advantage in practice over the other schemes.

other ANTLR and rdp have support for automating building an abstract syntax tree; this is helpful if not all processing can be done in one left-to-right pass and we have to build a tree on which we can run several passes and in any order. Rdp also supports running multiple passes on the source text, which may also help in such situations. There are also some other supporting features in the bigger generators, e.g., rdp has support for parsing the command line.

Looking at Fig. 3, we see that the backtracking parsers are much smaller than the others; the main reason for this is probably that the backtracking parsers do not need to analyse the grammar, in contrast to the other parser generators. Bnfparse and mop don't even need to generate a parser, because they interpret the grammar directly. DCG does not need to implement backtracking, because that is built into Prolog.

The other feature that is strongly correlated with size is error recovery. However, I expect that adding Coco/R-like error recovery to Gray will only add about 100–200 lines, so this cannot explain the difference in size we see, so we will take a closer look at the differences in the next section.

3 Gray vs. Coco/R

The parser generators closest in functionality to Gray are ell, Coco/R, and rdp (in this order). Coco/R is the smallest of these programs, so I compare Gray with Coco/R.

3.1 Reasons for size differences

Differences in the code size (as measured in source lines) can have several causes:

Functionality

In the present case, the most significant differences are the scanner and the error recovery present in Coco/R; other features present in Coco/R, but not in Gray are output options like listing generation, cross-reference list, statistics, debugging info, etc. Another difference in functionality is the presence of hard limits in Coco/R (it uses fixed-size arrays instead of dynamic memory allocation), whereas Gray is limited only by available memory.

Language-induced design decisions

The most significant differences in our example are:

- Gray uses *run-time code generation* (not available in Modula-2), Coco/R outputs source code for the parser. Run-time code generation makes many things simpler: e.g., there is no need to generate code for the sets; instead, the sets produced by the parser generator can be directly used in the parser.
- Gray uses the Forth parser for *reading the grammar*, Coco/R uses a separate parser (generated with Coco/R); this also implies that Coco/R has to do command-line and file handling. Modula-2 impedes a Gray-like solution in two ways: its restrictive syntax would make the grammar syntax relatively cumbersome; and its compile/run-time separation would make this solution cumbersome to use (first compile the grammar together with the parser generator, then run it before you get a parser).
- Some of the output options that I listed earlier as differences in functionality, may also be language-induced: The *compile/run-time separation* of Modula-2 makes it impossible for the Coco/R user to access the internal data structures of the parser generator, so the programmer of the parser generator provides ways to get information out of them; in contrast, such features are much less important to Forth programmers, because it is easy to add this functionality when it is needed (which may be never), and in exactly the way it is needed.

- Gray uses Forth's dictionary as *symbol table*, Coco/R has to implement its own symbol table. This is related to the issue of using the Forth parser for reading the grammar.

- The output language requires Coco/R to deal explicitly with *values passed between rules*, and with declarations. In contrast, with Forth the values are passed on the data stack, and Gray just has to avoid changing the stack (and that's easy).

- Gray implements and uses *object-oriented* techniques for dealing with the various grammar constructions. (The Modula-2 version of) Coco/R does not have this option, because Modula-2 is not an object-oriented language, and its static typechecker does not allow the programmer to add object-oriented features; as a result, Coco/R uses complex case-analysis code (total: 338 `IF`s, 30 `ELSIF`s, 18 `CASE`s), whereas Gray uses object-oriented dispatch to simple procedures (just 33 `if`s).

Other design decisions

Of the many design decisions done differently, two attracted my attention:

The data structures for the grammar differ in one important point: Gray uses a separate concat node to represent the sequence in the grammar, whereas Coco/R apparently treats the sequence as the default and has a *next* pointer in all grammar nodes. Looking at the source code, it appears that the Coco/R data structure requires additional code (there are procedures for starting and ending a sequence), but I would have to program both to be sure which one is better.

Coco/R tries to save space by having only one copy of each distinct set. Gray makes few efforts to save space (one exception is that `union` checks whether the result is equal to one of the inputs), but the overall space consumption is so small that such efforts would rarely pay for themselves: When generating an Oberon parser, only 2.2KB of sets are produced.

Coco/R uses a large number of global variables, Gray uses only five. It is not clear how this affects the code size.

Programming style

Gray is written in a relatively vertical programming style; e.g., an `if`...`else`...`endif` structure uses at least five lines. In contrast, Coco/R uses a more horizontal programming style; e.g., many `IF`-statements are written in one line, if they fit. I would expect Coco/R to have > 310 lines more if it was written in a more vertical style.

	Gray	Coco/R
code generator	65	700
command line, files	-	159
first-set, check conflicts	83	327
follow-set	26	95
grammar constructors	47	78
misc	38	504
objects	35	-
scanner	-	1300
sets	72	188
symbol table	-	274
syntax	24	258
types	83	122
total	473	4005

Figure 4: Code sizes of various subtasks

Language requirements

Modula-2 requires an additional declaration (in the definition module) of any publically visible procedure etc. It also requires the use of parameters that have to be declared where Forth just passes data on the stack.

Forth encourages breaking the program into many small (often only one line long) definitions, each of which has a one-line header (in the programming style used with Gray); this factoring can also have a positive effect on the line count by encouraging reuse.

3.2 Piecewise comparison

I do not understand Coco/R well enough to separate all of the code according to the issues discussed above, but I started by separating it into parts according to their function, which allows to quantify some of the effects discussed above.

code generator Only about one third of Coco/R's code generator deals with issues that also occur in Gray. The rest of the code does various I/O, copy files, etc. This is the consequence of generating source code, forced by Modula-2's lack of run-time code generation.

command line, files The necessity of this code in Coco/R is necessitated by running it as a standalone program and having its own parser for reading the grammar; and this is necessitated by the compile/run-time separation in Modula-2, and the unflexible syntax.

first-set, check conflicts This is the code for computing the fist sets, for reporting left-recursion, checking the LL(1) property etc. (this is in one category, because it is implemented as one pass in Gray). The main difference here is, again, the difference in the data structure.

follow-set This code computes the follow sets. The main difference is in the data structures.

grammar constructors This is the code that builds nodes for the internal representation of the grammar. Again, the main difference is in the data structures.

misc Everything that does not fit elsewhere, e.g., support words; for Coco/R this also includes some features that are not present in Gray (e.g., the cross-reference), as well as non-features (e.g., reporting overruns of the hard limits coded into Coco/R). For Coco/R it may also contain stuff that supports just one other piece of code and should be categorized there.

objects This is a simple structures and objects package; more general and elaborate versions were developed later [Ert97c, Ert97b].

scanner The scanner generator included in Coco/R, including the parts of the parser concerned with the syntax of the scanner generator.

sets The sets package (used in various ways in the parser generators). One difference here is that Coco/R's sets are used with destructive updates (`Unite`), whereas Gray uses functional operators (`union`). Another difference is that Coco/R uses fixed-size sets, whereas Gray's user can set the set size according to his needs.

symbol table The code in Coco/R that implements the symbol table.

syntax For Gray, this is the code implementing the syntactic sugar for the sequence and alternative grammar constructors. For Coco/R, this is the grammar of Coco/R, including the actions, but without the parts needed for the scanner generator.

types These are structure definitions and method maps for Gray; type, constant, and global variable declarations for Coco/R.

One third of Coco/R's code is for the scanner, a major difference in functionality. The other major difference in functionality is the error recovery, but I have not separated the code for this out (I actually found very little, probably because it is mixed with the rest).

1178 source lines of Coco/R are due to language-induced design decisions (most of the code generator; command line, files; symbol table; syntax). Even if we left out the additional features of Coco/R and if the rest of the code was just as large as Gray's code, this issue alone would make Coco/R more than three times larger than Gray. This is the strongest indication we have in this comparison that using Forth results in significantly smaller code than using an Algol-family language like Modula-2.

3.3 Comparison of similar functions

We see a factor of 2–4 in favour of Gray for the most comparable parts: first-set, check conflicts; follow-set; grammar constructors; sets. This difference could be caused by the differences in design (e.g., the data structure), and/or by some inherent code density advantage of Forth over Modula-2.

To investigate this issue, I looked at the *sets* code. Here the data structures are quite similar, except that Coco/R uses a fixed-size array to allocate its sets, whereas Gray uses dynamic allocation. The general style of the operations differs: Gray uses constructive operations (`union`), whereas Coco/R uses modifying operations (`Unite`).

Coco/R has more specific operations (`Fill`, `Excl`, `Elements`, `Empty`, `Equal`, `Differ`, and two printing operations); Gray has fewer specific (only `singleton` is not present in Coco/R), but a few building-block operations (`apply-to-members`, `binary-set-operation`, `binary-set-test?`), that are used to build the specific operations and can be used to build more. The differences in the implemented specific operations is apparently caused by differences in higher-level design; e.g., Gray does not have a set difference operation, because it does not need it.

I looked at the common operations of the sets code (which includes the building-blocks `binary-set-operation` and `binary-set-test?` for Gray). They have 61 lines in Coco/R and 49 lines in Gray. This can be explained with the small inherent code density disadvantage of Modula-2, which needs more declarations.

In Fig. 5 and 6 you can see the two implementations of set union. One reason for using a general binary-set-operation in Forth and not in Modula-2 is that you can pass the execution token of `or` in Forth, but you cannot pass the BITSET-+ as procedure (you would have to write a wrapper procedure and pass that). The non-use of Modula-2's FOR in Fig. 5 is probably due to the origin of Coco/R in Oberon, which has no FOR.

There appears to be no significant inherent code density advantage of Forth over Modula-2 at the basic coding level. The size differences appear to be caused by differences in design; this suggests the question of whether the language used plays any role in these design decisions, but I will leave that to further work.

```
PROCEDURE Unite (VAR s1, s2: ARRAY OF BITSET);             (* s1 := s1 + s2 *)
(* Unite                s1 := s1 + s2
----------------------------------------------------------------------------*)
PROCEDURE Unite (VAR s1, s2: ARRAY OF BITSET);
  VAR
    i: CARDINAL;
  BEGIN
    i := 0; WHILE i <= HIGH(s1) DO s1[i] := s1[i] + s2[i]; INC(i) END
  END Unite;
```

Figure 5: Set union in Coco/R (from definition and implementation module).

```
: binary-set-operation ( set1 set2 [w1 w2 -- w3] -- set )
\ creates set from set1 and set2 by applying [w1 w2 -- w3] on members )
\ e.g. ' or binary-set-operation  is the union operation )
 here >r
 cells/set @ 0 do >r
  over @ over @ r@ execute ,
  cell+ swap cell+ swap
 r> loop
 drop 2drop r> ;

: union1 \ set1 set2 -- set )
 ['] or binary-set-operation ;
```

Figure 6: Set union in Gray

4 Conclusion

The size advantages of the Forth programs are impressive. A part of this advantage is due to having less features.[3] But another, significant part is due to features present in Forth, and absent in Algol-like languages: run-time code generation, the Forth parser and its flexible syntax, the dictionary, the absence of static type-checking, and the programmer-visible stack. The lack of these features causes design decisions that require more than 1000 lines in Coco/R (more than twice the size of Gray).

Further work should investigate if this result is specific to this problem class, or also applicable to others. Moreover, it would be interesting to see if the language also plays a role in design decisions that led to the inclusion of more set operations and thus the size difference in the sets code. Finally, comparing programs by metrics other than the source code size is also an interesting topic.

Acknowledgements

Manfred Brockhaus provided valuable comments on this paper.

[3]Depending on the point of view (and on the current needs), this is a virtue or a shortcoming.

References

[EM95] M. Anton Ertl and Martin Maierhofer. Translating Forth to efficient C. In *EuroForth '95 Conference Proceedings*, Schloß Dagstuhl, Germany, 1995.

[Ert97a] M. Anton Ertl. GRAY – ein Generator für rekursiv absteigende Ybersetzer. In *Forth-Tagung*, Ludwigshafen, 1997. In German.

[Ert97b] M. Anton Ertl. Yet another Forth objects package. *Forth Dimensions*, 19(2):37–43, 1997.

[Ert97c] M. Anton Ertl. Yet another Forth structures package. *Forth Dimensions*, 19(3):13–16, 1997.

[Han90] Hanspeter Mössenböck. A generator for production-quality compilers. In *Compiler Compilers*, volume 477 of *LNCS*, pages 48–61. Springer, 1990.

[IH90] Tyler A. Ivanco and Geoffry Hunter. A user definable language interface for Forth. *Journal of Forth Application and Research*, 6(1), 1990.

[Rod90] Brad Rodriguez. A BNF parser in Forth. *SigForth Newsletter*, 2(2):13–15, December 1990.