

Convert on the MC68000: A Compiled String-Processing Language

Gerardo Cisneros*

Departamento de Aplicación de Microcomputadoras,
Instituto de Ciencias, Universidad Autónoma de Puebla,
Apartado postal 461, 72000 Puebla, Puebla, México.
BITNET: CISNEROS@UNAMVM1

October 11, 1991

Abstract

String-processing languages have typically been implemented as interpreters, macro generators or program generators. Convert, a language whose main appeal is the ease with which string transformations may be expressed as sets of pattern-matching and substitution rules, was developed initially as an interpreter under a symbol manipulation version of REC. This paper describes the implementation of a machine-code compiler of Convert for the Motorola MC680x0 family of microprocessors. The execution speed of programs produced with this compiler compares favorably with that of equivalent programs written in other string-processing languages.

Introduction

Languages specializing in string processing [Ah79, Gr72, Gr80, Ci85a] (which in a broader sense include lexical scanner and parser generators [Jo75, Le75]) have been for the most part implemented as interpreters, macro generators or program generators. Except for lexical scanner and parser generators, whose narrow scope of applicability to regular expressions and context free grammars, respectively, allows for efficient programs to be generated, most of the existing string-processing languages exhibit slow execution speeds. This is usually considered a reasonable tradeoff for the ease of programming these languages provide.

This paper discusses the implementation of a compiler for the string-processing, pattern-matching and substitution language Convert which generates machine code for the Motorola MC680x0 family of microprocessors. The compiler is written in Convert itself and generates fast, compact programs which compare favorably in size and speed with programs written in other languages such as Icon and Awk.

1 An Overview of Convert

Convert [Ci85a, McI86] is the string-oriented version of CONVERT [Gu65, Gu66], a pattern-matching and substitution language modeled after LISP and COMIT. Convert's control structure is derived from Markov algorithms and its use of variables in pattern-matching and substitution from Post production systems.

*On sabbatical leave for the academic year 1991–92 at Departamento de Innovación, Dirección General de Servicios de Cómputo Académico, Universidad Nacional Autónoma de México, Apartado postal 20-059, 01000 México, D.F., México

A Convert program is a quadruple formed by a list of pattern definitions, a list of skeleton definitions, a list of variables and a rule set. Each rule in the rule set is formed by a pattern and a skeleton separated by a comma and enclosed in parentheses. Rules may be repetitive or terminal: a rule followed by a colon is repetitive; a rule followed by a semicolon is terminal. The list of variables (which are denoted by small integers) is a declaration of those variables which are unbound before each rule's pattern-matching attempt, and which may become bound during the pattern-matching process.

Patterns and skeletons anywhere in the quadruple may refer to definitions in the first two lists of the quadruple or to any definitions inherited by dynamical scoping. Functions are defined by naming quadruples, which can then be referenced as skeletons.

Given a string as an argument, a Convert program transforms the string by means of its rule set. The pattern in each rule is matched in turn against the text. If a match succeeds, the corresponding skeleton generates the transformed text and the type of rule determines whether to terminate (returning the new text as the program's value) or to iterate the rule set on the new text, beginning with the first rule. If no pattern matches the string is returned unaltered. In a successful match, variables which became bound may be used on the skeleton side of the rule in the production of the new string.

Convert has a very general set of pattern forms including constants, variables, intervals, minimal and maximal iterations and Boolean combinations. This permits concise descriptions of fairly complex strings or collections of strings.

A simple example of a pair of Convert functions to remove tabs from a string, expanding them with blanks to the customary 8 columns, follows.

```
[find all tabs, call fill to expand them]
((())(0 1 2)(
  [rule 1: a tab within the first 8 characters]
  [<-----pattern-----> <-----skeleton----->]
  ((and,<[8]>,<0>(^I)<1><2>,(fill,<0>)(xtab,<1><2>));

  [rule 2: no tab within the first 8 characters]
  ((and,<[8]>,<0><2>,<0>(xtab,<2>));

  [rule 3: less than 8 characters, at least one tab]
  (<0>(^I)<1>,(fill,<0>)(xtab,<1>));
)) xtab

[fill with blanks to eight characters]
((())(0)(
  [rule 1: truncate text to the first 8 characters]
  ((and,<[8]>,<0>,<0>);

  [rule 2: append 8 blanks to the current text, repeat]
  (,<=>      ):
)) fill
```

Comments, enclosed in brackets, are allowed anywhere but within a pattern, a skeleton or a variable list. In rule 1 of `xtab`, (`and,...`) is the Boolean **and** of the patterns it lists. Each of the patterns after the first listed by an (`and,...`) must match exactly the same text matched by the first pattern. `<[8]>` is a pattern matching any 8-byte string. `<0>`, `<1>` and `<2>` are variables 0, 1 and 2, respectively, which by appearing in the list of variables are unbound at the beginning of each rule's attempt at pattern matching. `(^I)` is a constant pattern matching the horizontal tab (HT) ASCII character. `(fill,<0>)` is a skeleton which invokes function `fill` with a argument equal to the value bound to variable 0, and `(xtab,<1><2>)` is a recursive invocation of `xtab` on the concatenation of the values bound to variables 1 and 2.

The pattern in a rule is not in general required to match the target text in its entirety; many times it is only a prefix that one is looking for. A specific pattern is provided which will only match the null string at the end of the text. On the other hand, an unbound variable at the end of a pattern (as in all three rules in function `xtab`, or in the second pattern of all instances of `(and, . . .)` shown above) is guaranteed to match (and bind) all of the remaining text. In the case of the `(and, . . .)` patterns, the “remaining text” is constrained by the text matched by the first pattern in the list.

In rule 2 of `fill`, the pattern is empty, which guarantees a match no matter what the text is (all strings have a null prefix). `<=>` is a skeleton whose value is the original text coming into the (successful) rule, and the eight blanks between `<=>` and the right parenthesis form a constant skeleton which produces eight blanks.

Complete tables of pattern and skeleton forms appear in [McI86].

Convert has been available in public domain libraries for some time [McI85a, Ci86], implemented as a program generator based on REC [McI68, McI86], a concise language reminiscent of APL and FORTH. In fact, the development of REC was guided by the desire to have a language into which Convert could be easily compiled. The processor-operating system combinations for which REC and Convert are available include the following: Intel 8080-CP/M; Intel 8088-CP/M; 8088-MS-DOS; Motorola MC680x0-UNIX, MC68000-AMOS. This Convert compiler, written in REC, generates REC programs which are then run by the REC load-and-go compiler.

Several applications have been developed, among others

- Symbolic calculation of matrix elements for a quantum-chemical application [Ci85b].
- Knight’s tour [McI85b], LIFE on a torus [McI85c] and linear cellular automaton evolution.
- An instructional package for editing automata (regular expressions, pushdown automata, Turing machines, Markov algorithms, Post systems), compiling them into REC programs, and executing them [Ci89].

As these evolved, questions of speed and efficiency became apparent. Cellular automaton evolution tended to run a thousand times slower than specially constructed machine language programs. Parsing was less extreme; although conciseness and clarity of expression are definite advantages of Convert, a parser for PL/0 [Wi76] written in Lex [Le75] and Yacc [Jo75] is almost ten times faster. (It should be kept in mind, however, that Lex and Yacc are strictly confined to regular expressions and LALR(1) parsers, respectively, whereas Convert was designed for the Turing machine level of complexity.)

There are two obvious ways to make Convert faster. The first one is to improve the underlying REC processor to minimize string movements among its data structures (the program area, the pushdown list and the workspace); profiling showed this to be the principal source of wasted activity in the parser application. The second is to use Convert to compile itself into machine code, which is similar to the approach used to improve the automaton evolution applications—take advantage of Convert’s transformation capabilities to translate a description of neighborhoods and rules of evolution into machine language programs capable of achieving the best possible speed of the given machine.

Since Convert provides a very natural medium in which to express symbolic transformations, it should make a good language for writing compilers. The first step in compiling Convert into other than REC was to write a Convert compiler for itself. To ensure consistency, this compiler was set up first to generate REC code. The conversion to native-code generation was then just a question of replacing the REC-generating skeletons for skeletons producing assembler code, and rewriting the run-time library (which in the REC version is also written in REC and is read by a run-time start-off routine inserted by the Convert compiler).

2 New language features

Compiling Convert in Convert allowed us to add features not present in the REC-based compiler reported in [McI86]:

- Long names for pattern and skeleton definitions and for functions—the common convention of a letter followed by none or more letters or digits is now allowed.
- A new convention for statically *vs.* dynamically scoped procedure names (which include pattern, skeleton, and functions definitions): names starting with an uppercase letter are dynamically scoped; names starting with a lowercase letter are statically scoped.
- Separate compilation. A Convert program not containing a main function (i.e., an unnamed quadruple just before the end) acquires a global name derived from the source file name by dropping its extension and prefixing it with an underscore; execution of a skeleton with this name invokes the last function of the corresponding file.

A feature removed from both the machine-code and REC versions of Convert is the $(\text{LAM}, (\omega), \sigma)$ skeleton, which provided a set ω of unbound local variables for use in skeleton σ . This skeleton was used only in connection with the pattern-directed file-reading skeleton; the syntax of the pattern-directed reading skeleton was modified to allow declaring variables to be used in pattern matching.

3 Implementation

Convert requires two main structures: a workspace, which contains the text currently being transformed, and a table of pointers. Each pointer is associated with a variable and either indicates the variable has not been declared or points to the variable's current descriptor (address and length) contained in a stack frame, thus allowing dynamical scoping of variable bindings.

The Motorola MC680x0 CPU provides a large enough set of registers that pointers to the main structures can be kept in them, and the addressing modes make it easy to work with these pointers. Register A6 is used as a frame pointer. Stack frames are created by structures with variable declarations (even if the declaration lists no variables); they contain a descriptor for each variable declared. Each descriptor in a frame determines whether the corresponding variable has been bound, and to what. Registers A2 through A5 are used as workspace pointers: A2 and A5 delimit the workspace text, A3 points to the beginning of the substring to be considered next and A4 points to the end of the substring to be considered (it moves only when the Boolean **and** is involved). Register A1 points at the array of pointers to the stack frame variable descriptors. Register D7 contains the number of variable descriptors in the current frame. Registers A0 and D0 through D6 are used as general-purpose registers.

The implementation of skeletons and of the simpler pattern forms is quite straightforward; compilation of more complex patterns is guided by the requirement that the concatenation of two patterns, $\pi_1\pi_2$, match a string τ if there exists a prefix τ_1 and a suffix τ_2 such that $\tau = \tau_1\tau_2$, π_1 matches τ_1 and π_2 matches τ_2 . If more than one partition of τ exists satisfying the conditions, the one with the shortest τ_1 is chosen. (Icon provides an alternative to this choice by means of *generators* [Gr80], which return each possible match in a sequence of references.)

When π_1 is capable of matching more than one substring, π_1 and π_2 are compiled into a loop that tries each possible match for π_1 (in the appropriate order) until π_2 matches too; the combined pattern succeeds if π_2 matches and fails if alternatives for π_1 are exhausted.

Patterns requiring such provisions for backtracking include variables, variable-length intervals, minimal iterations, Boolean combinations, and, as a compiler option, patterns in definitions and references to them. These are handled as follows:

- A variable in non-final contexts compiles the pattern following it into a subroutine whose address is passed as an argument to the library function handling variable binding. The library function called depends on whether the remaining pattern is just a literal, a literal followed by some other pattern, or a pattern not starting in a literal.
- A variable length interval (pattern $\leftarrow\rightarrow$) is compiled into a loop which increases the interval from zero length until the remaining pattern matches or the available text is exhausted. The interval length is increased one byte at a time unless the remaining pattern starts with a literal,

in which case successive searches for the literal govern how the variable interval is matched each time through the loop.

- In a Boolean conjunction ($\text{AND}, \pi_1, \pi_2, \dots, \pi_n) \pi'$, patterns $\pi_2, \dots, \pi_n$ and π' are compiled as a subroutine in which all of $\pi_2, \dots, \pi_n$ must first match the same string matched by π_1 and then π' is tested. The code generated for π_1 calls the subroutine in a context appropriate to the nature of π_1 .
- In Boolean disjunctions ($\text{OR}, \pi_1, \pi_2, \dots, \pi_n) \pi'$, π' is compiled as a subroutine to be invoked by each of the alternatives $\pi_1, \dots, \pi_n$ in a context appropriate to its nature. The subroutine's address is stored in a stack located below the array of pointers to the variable descriptors; nesting of ORs or references to patterns in definitions in the backtracking option cause the generation of calls to library functions to manipulate this stack and make calls through it.
- Pattern definitions in the backtracking option are compiled as if they had a trailing pattern, which gets compiled as a subroutine whose address is obtained from the run-time stack lying below the pointer array. References to a pattern in a definition compile their actual trailing pattern (even if null) into a subroutine whose address is pushed onto the stack before the pattern in the definition is invoked.

4 Library

The compiler is supported by a library containing 52 modules, most of them coded in 68000 assembly language.

Modules are included for:

- Pattern matching: string searches and comparisons, variable matching in four different contexts (final variable, variable followed by final constant, variable followed by non-final constant, variable followed by non-constant), lexicographic interval testing, OR and pattern definition call stack control, parenthesis nest, debugging patterns PWS (print remaining workspace yet to be matched) and PVR (print variable). One auxiliary module unbinds variables in the current scope and another contains entry points to save and restore unbound variables in the current scope.
- Skeletons: insert a constant string, insert a copy of the most recent argument, convert an integer to ASCII and insert it.
- Built-in functions: arithmetic operations and numeric conversions, character string conversions, file and pseudofile operations (opening, closing, reading, writing, linking, renaming, status inquiry), process control (process, group and user id inquiry; command execution)
- Auxiliary: saving the working storage, trapping illegal use of variables, control of file buffers, delimiting function arguments, retrieving integer arguments.

5 Results

The machine-code compiler produces code which speeds up Convert applications by about an order of magnitude, rendering Convert competitive with other specialized languages. Table 1 shows the results for a program which determines all congruence relations of a deterministic finite automaton (adapted to string Convert and Icon from [McI69]) and Table 2 shows the results for a topological sort program [Ah88, Gr83].

As far as other applications are concerned, the PL/0 parser is only 25% slower than the corresponding Lex/Yacc version (both having comparable code sizes) and 7 times faster than the REC/Convert version; the quantum chemical application HAMEL [Ci85b] (which is not amenable to treatment by Yacc) is 7 times faster than the REC/Convert version when intermediate files are

	Convert (native)	Convert (REC)	Icon (lists)	Icon (strings)
Dual 83/80 (68000, 8MHz)	41.6	578	169	562
PC clone (8088, 10MHz)	92.8	488	240	828
NeXTstation (68040, 25MHz)	1.9	N/A	6.1	20.9

Table 1: Timings in seconds for the calculation of all congruence relations of an 8-state finite automaton. The native-code Convert for PC's is that of [Ca89].

	Convert	Awk	Icon	tsort
NeXTstation (68040, 25MHz)	62.8	2.4	2.0	0.2

Table 2: Timings in seconds for a topological sorting of a graph with 75 nodes and 1100 edges. `tsort` is the standard UNIX utility

written (as in the original program) and 12 times faster when pipes are used. An algorithm for hyphenating Spanish words which can be implemented using just Lex [Ma87] is about half the size and only a factor of two slower when programmed in Convert. A filter for a library application originally programmed in Awk runs six times faster when recast in Convert. Finally, a program to count lines, words and characters read from the standard input runs at about a fifth of the speed of the corresponding C program, which is not bad considering arithmetic is not one of Convert's strengths.

Directions into which Convert may be profitably extended are:

- Adding types: allowing patterns to match structures other than strings, e.g., lists, trees, etc.
- Adding associative arrays (a strength of Awk, as is evident from Table 2.)
- Improving the efficiency of arithmetic.

Nevertheless, Convert as it now stands is a viable alternative to string processing for applications which are best or most easily expressed in terms of rules of transformation.

Acknowledgments

The work reported herein was supported in part by grant #P228CCOX904651 from the Consejo Nacional de Ciencia y Tecnología and grant #C90-01-0473 from the Dirección General de Investigación Científica y Superación Académica of the Secretaría de Educación Pública. The author is a member of the Sistema Nacional de Investigadores.

References

- [Ah79] A. V. Aho, B. W. Kernighan and P. Weinberger, "AWK—a pattern scanning and processing language", *Software—Practice and Experience* **9**, (1979) 267–280.
- [Ah88] A. V. Aho, B. W. Kernighan and P. Weinberger, "The AWK Programming Language," Addison-Wesley, Reading, Mass. (1988)
- [Ca89] J. Ariel Carrasco-Ochoa, Lucía A. de la Vega-Doria y Gerardo Cisneros, "Compilador de Convert para el microprocesador Intel 8086," Proceedings of the Fifth International Conference *Computers in Institutions of Education and Research* (México City: Universidad Nacional Autónoma de México and Unisys, in press)

- [Ci85a] G. Cisneros y H. V. McIntosh, “Introducción al lenguaje de programación Convert,” *Acta Mex. Cienc. Tecnol.* **3**, 9 (Jan-Mar 1985) 65–74. For an English translation, see *SIGPLAN Notices* **21**, 4 (April 1986) 48–57.
- [Ci85b] G. Cisneros, C. F. Bunge and H. V. McIntosh, “Automatic generation of configuration interaction Hamiltonian matrix elements,” *Int. J. Quant. Chem. Symp.* **18** (1985) 683–689.
- [Ci86] G. Cisneros and H. V. McIntosh, REC and Convert compilers for MS-DOS, PC/Blue library Vols. 211 and 212, New York Amateur Computer Club (1986); also available as disk #MS29 from Micro Cornucopia.
- [Ci89] Gerardo Cisneros y Harold V. McIntosh, “REC y Convert en la enseñanza de la Teoría de Autómatas,” Proceedings of the Fifth International Conference *Computers in Institutions of Education and Research* (México City: Universidad Nacional Autónoma de México and Unisys, in press)
- [Gr72] R. E. Griswold, “The Macro Implementation of SNOBOL4, A Case Study in Machine-Independent Software Development,” W.H. Freeman, San Francisco (1972)
- [Gr80] R. E. Griswold and D.R. Hanson, “An alternative to the use of patterns in string processing,” *ACM Trans. Prog. Lang. Syst.* **2** 2 (1980) 153–172.
- [Gr83] R. E. Griswold and M. T. Griswold, “The Icon Programming Language,” Prentice-Hall, Englewood Cliffs, NJ (1983)
- [Gu65] A. Guzmán, “CONVERT,” B.S.E.E. Thesis (Spanish) Instituto Politécnico Nacional, México (1965)
- [Gu66] A. Guzmán and H. V. McIntosh, “CONVERT,” *Commun. ACM* **9**, 8 (August 1966) 604–615.
- [Jo75] S. C. Johnson, “Yacc—Yet Another Compiler-Compiler,” Comp. Sci. Tech. Rep. No. 32, Bell Laboratories (July 1975)
- [Le75] M. E. Lesk, “Lex—A lexical analyzer generator,” Comp. Sci. Tech. Rep. No. 39, Bell Laboratories (October 1975)
- [Ma87] J. A. Mañas, “Word division in Spanish,” *Commun. ACM* **30**, 7 (July 1987) 612–616.
- [McI68] H. V. McIntosh, “A CONVERT compiler of REC for the PDP-8,” *Acta Mex. Cienc. Tecnol.* **2**, 1 (Jan-Apr 1968) 33–43.
- [McI69] H. V. McIntosh, “CGRU and CONG: CONVERT and LISP programs to find the congruence relations of a finite state machine”, *Acta Mex. Cienc. Tecnol.* **3**, 1 (Jan-Apr 1969) 32–45.
- [McI85a] H. V. McIntosh and G. Cisneros, REC and Convert compilers for CP/M, SIG/M Vols. 213, 214 and 215, Amateur Computer Group of New Jersey (Jan 1985)
- [McI85b] H. V. McIntosh, Convert programming—MAZE and TOUR, SIG/M Vol. 245, Amateur Computer Group of New Jersey (Sept 1985)
- [McI85c] H. V. McIntosh, Convert programming—LIFE (Toru), SIG/M Vol. 246, Amateur Computer Group of New Jersey (Sept 1985)
- [McI86] H. V. McIntosh and G. Cisneros, “The programming languages REC and Convert,” *SIGPLAN Notices* **25**, 7 (July 1990) 81–94.
- [Wi76] N. Wirth, “Algorithms + Data Structures = Programs,” Prentice-Hall, Englewood Cliffs, N.J. (1976) pp. 307–310.