A Native Code Compiler of Convert for the
MC68000

Gerardo Cisneros
Departamento de Aplicación de Microcomputadoras,
Instituto de Ciencias, Universidad Autónoma de Puebla,
Apartado postal 461, 72000 Puebla, Puebla, México.

The purpose of this paper is to describe a machine code compiler of Convert
and some of the more interesting compilation problems posed by the semantics of
the language. Convert [Ci85a, McI86], is the string-oriented version of CONVERT
[Gu65, Gu66], a pattern-matching and substitution language modeled after LISP,
COMIT, Markov algorithms and Post production systems.

Convert’s main feature is that it has a very general set of pattern forms including
constants, variables, intervals, minimal and maximal iterations and Boolean com-
binations. This permits concise descriptions of fairly complex strings or collections
of strings, but raises also a few implementation problems.

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 either repetitive or terminal and is formed in turn by a pattern and a skeleton,
either of which may refer to definitions in the first two lists of the quadruple. The
list of variables is a declaration of those variables which are considered unbound at
the start of each rule, and which may become bound during the pattern-matching
process.

Given a string as an argument, a Convert program acts to transform the string
by means of its rule set. The pattern in the first rule is matched against the text.
If the 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 the match fails, the next rule in turn is tried, and so on until there is
a match or no more rules remain, in which case the text is returned unaltered.

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

[find all tabs, call fill to expand them]
xtab ((())(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>));
)

[fill with blanks to eight characters]
fill (()())(0)

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

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

Terminal rules are followed by a semicolon, repetitive rules are followed by a
colon. Comments, enclosed in brackets, are allowed anywhere but within a pattern,
a skeleton or a variable list. 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 rule
(as in all three rules in subroutine xtab) is guaranteed to match (and bind) all of
the remaining text.

The central problem in the compilation of Convert is posed by the requirement
that the concatenation of two patterns, $\pi_1\pi_2$, matches a string $\tau$ if there exists a
prefix $\tau_1$ and a suffix $\tau_2$ such that $\tau = \tau_1 \tau_2$, $\pi_1$ matches $\tau_1$ and $\pi_2$ matches $\tau_2$. If more than one partition of $\tau$ exists satisfying the conditions, the one with the shortest $\tau_1$ is chosen.

In the most general case, $\pi_1$ will be some pattern that may match more than one substring. Then $\pi_2$ must become a subroutine, to be invoked by the procedure examining alternatives for $\pi_1$: each time a possible match for $\pi_1$ is chosen, $\pi_2$ must be matched against the remaining substring; if it does not match a reconsideration of $\pi_1$ is necessary. Patterns $\pi_1$ requiring this treatment include variables, Boolean combinations, references to pattern definitions, and minimal iterations.

A functional compiler for Convert has been available in public domain libraries for some time [McI85a, Ci86], based on the REC language [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-machine combinations for which REC and Convert are available include the following: Intel 8080–CP/M; Intel 8088–CP/M; 8088–MS-DOS; Motorola MC68000–UNIX, MC68000–AMOS. This Convert compiler 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 [McI85e] and linear cellular automaton evolution.
- PL/0 [Wi76] parser for instructional purposes [McI86].

As these evolved, questions of speed and efficiency became apparent. 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, the well-known combination of lex [Le75] and yacc [Jo75] turned out to be almost ten times faster when applied to PL/0. (It should be kept in mind, however, that lex and yacc are strictly confined to regular expressions and LALR(1) grammars, respectively, whereas Convert was designed for the Turing machine level of complexity.)

Two ways to make Convert faster became apparent: The first one was to improve the underlying REC processor to minimize string movements among its data structures (the program area, the pushdown list and the workspace) which profiling
showed to be the principal source of wasted activity in the parser application. The second was 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. Thus 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. However, 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), a process which took about three months of the author’s time.

The Motorola MC68000 CPU provides a large enough set of registers that pointers to the main structures (the workspace and the table of pointers to variables) 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 and determines which unbound variables may be bound in a pattern. 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 an array of pointers to the variable descriptors (which reside in stack frames). Register D7 contains the number of variable descriptors in the current frame. Registers A0 and D0 through D6 are used as general-purpose registers.

As an example, consider the pattern $(itr, \pi)\pi'$. The pattern $itr$ matches the minimal iteration of instances of $\pi$ such that $\pi'$ can be made to match. Thus $\pi$ will be tested only as long as $\pi'$ has not matched, and the entire pattern will fail if $\pi$ ever fails. Compilation of this pattern then produces the following:

$$L\alpha: \text{jsr svubv}_{-} \quad |\text{save list of unbound, bindable variables}|$$
$$\text{movl a3,sp@-} \quad |\text{save current substring pointer}|$$
$$\ldots \quad |\text{code for } \pi' \text{ goes here; it will contain}|$$
$$\ldots \quad |\text{a jump to L}/3 \text{ for the false case}|$$
$$\text{addql #4,sp} \quad |\text{true; get rid of saved pointer}|$$
$$\text{movw sp@+,d0} \quad |\text{get number of items saved by svubv}|$$
A counter, a short array and a stack are used to generate and keep the needed labels straight; compilation proceeds by recursive descent, as suggested by the example.

The compilation of other interesting patterns like OR and variables will be shown in the full paper.

The resulting compiler produces code which speeds up Convert applications by about an order of magnitude; for instance, the PL/0 parser is only 25% slower than the corresponding lex/yacc version and 7 times faster than the REC/Convert version; the quantum chemical application HAMEL [Ci85b] is 7 times faster than the REC/Convert version when intermediate files are written (as in the original program) and 12 times faster when pipes are used.

The native-code compiler provides now an alternative way to port Convert to other machines. For machines with instruction sets similar to the 68000's (e.g., VAX), a translator may be easily written in Convert to port the native-code compiler directly. For machines with very different architectures, porting REC first and building on it (as has been done in the past) will remain a viable alternative.

References


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.


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)

H.V. McIntosh, Convert programming—MAZE and TOUR, SIG/M Vol. 245, Amateur Computer Group of New Jersey (Sept 1985)

H.V. McIntosh, Convert programming—LIFE (Toru), SIG/M Vol. 246, Amateur Computer Group of New Jersey (Sept 1985)
