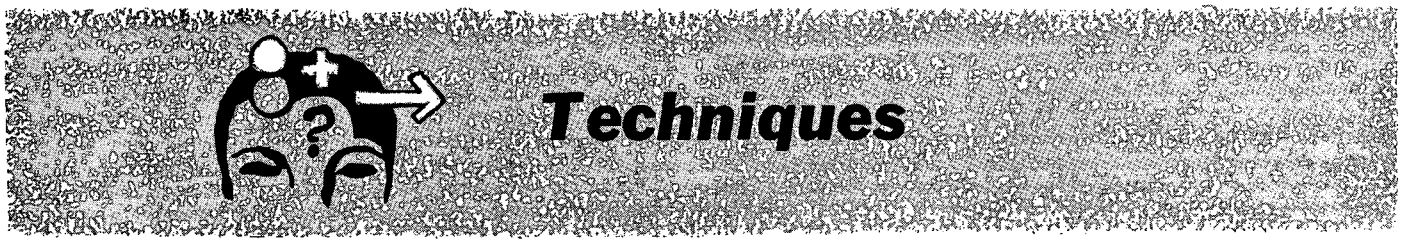

C. L. McCARTY, JR., Editor

Peephole Optimization

W. M. McKEEMAN
Stanford University, Stanford, California

Redundant instructions may be discarded during the final stage of compilation by using a simple optimizing technique called peephole optimization. The method is described and examples are given.

Nearly all compilers expend some effort to increase the efficiency of the code they produce. One particularly simple and often neglected technique might be called *peephole optimization*, a method which consists of a local inspection of the object code to identify and modify inefficient sequences of instructions. Several examples are given below. The source language is ALGOL and the object code is given in a simple single-address assembly language; nevertheless, the technique is directly applicable to a wide class of languages and machines.

Example 1. Source code:

$$X := Y;$$
$$Z := X + Z$$

Compiled code:

```
LDA Y   load the accumulator from Y
STA X   store the accumulator in X
LDA X   load the accumulator from X
ADD Z   add the contents of Z
STA Z   store the accumulator in Z
```

If the store instruction is nondestructive, the third instruction is redundant and may be discarded. The action may be accomplished by having the subroutine which emits object code check every time it produces an LDA to see if the previous instruction was the corresponding STA and behave accordingly.

Example 2. Source code:

$$X := Y;$$
$$Z := Z + X$$

This example presents a problem which occurs repeatedly during the translation of arithmetic expressions. A commutative operator, "+", appears in the source code, but straightforward translation leads to less than optimum code. When the optimizing code emitter produces a commutative operator (addition, multiplication, transfer on equality or inequality, logical AND, logical OR, etc.), it must first check the preceding instruction. If that instruction was LDA then the optimizer may choose, if the instruction preceding the LDA was STA, to reorder the commutative operation so as to avoid the LDA as in Example 1. Having avoided the LDA, the optimizer may check to see if the STA was in fact a store into a temporary location in which case the STA may also be discarded.

Example 3. Source code:

$X := 2.0 \times 3.14159265/360.0 + X;$
if $X < 0.0$ **then** · · · · ·

Object code:

```
LDA = 2.0          load the accumulator with the constant
                     2.0
MUL = 3.14159265   multiply by pi
DIV = 360.0        divide by 360.0
ADD X              add the contents of X
STA X              store the accumulator in X
LDA X              load the accumulator from X
SUB = 0.0          subtract the constant 0.0
TPA ···            transfer on positive accumulator ···
```

A programmer is often tempted to write an expression involving several constants simply because he does not have the time or facilities to compute the value by hand. If the optimizer, when it produces an arithmetic operation, checks to see if the operation addresses a constant and the preceding instruction is an LDA addressing a constant, then it may simplify the pair to a single LDA addressing the result of the operation on the constants. The optimizer may also discard such nonsense as addition or subtraction of zero and multiplication or division by one. The object code in Example 3 will reduce to:

```
LDA = 0.017453293
ADD X
STA X
TPA ···
```

Many machines have operations that are convenient in hand coding but difficult to fit into the more general structure of a compiler. Frequently the optimizer can be set to recognize a sequence of instructions as the equivalent of a hardware command and make an appropriate modification in the object code.

There are, however, situations in which the compiled code may not be improved.

This work was supported in part by the Office of Naval Research, Contract Nonr. 225(37).

Example 4. Source code:

$$X := Y;$$

$$L: \quad Z := X + Z$$

Even though the object code produced is identical to that of Example 1, the instruction LDA X can be reached via a transfer to the label L and must not be omitted. Similar considerations might be necessary for parameters that cause side effects through the evaluation of a function designator.

These techniques have been used in GOGOL, a translator written by the author for the PDP-1 time sharing system at Stanford. The optimizing code emitting routine consists of about 400 of the 2000 instructions required for the compiler. We have found that the code emitter can see, through a very narrow peephole, enough to make considerable improvement in the object code. The limitation on how well the optimizer will work seems to depend primarily on how much time and space are available for recognizing redundant sequences of instructions.

RECEIVED FEBRUARY, 1965

Method for Hyphenating at the End of a Printed Line

R. P. RICH AND A. G. STONE
Applied Physics Laboratory, Johns Hopkins University, Silver Spring, Md.

A description of a method of hyphenation is presented as a result of application of several general rules. The character sets considered by the routine and the method are briefly outlined.

Introduction

A simple set of rules for hyphenating words at the end of a line has proved to give reasonably good results:

(1) At least one vowel other than final E, ES or ED must be carried over to the new line.

(2) At least one vowel must be left on the current line.

(3) The new line cannot begin with a vowel or double consonant.

(4) No break is made between letters of some special pairs.

Most of the programming effort is required, not for breaking within words, but for handling nonalphabetic strings. Hence the following fuller discussion of the actual hyphenation routine may be of interest.

If any reader can find another rule or two general enough to make a statistically substantial improvement in the results we will be most happy to hear from him.

Preliminaries

The routine determines the breaking point of a line consisting of at most N characters. The breaking point after the Nth character is considered to be the *nominal break*. The routine considers β characters before and α characters after the nominal break (where α and β are given parameters). The *right margin* is the α characters after the nominal break (See Figure 1).

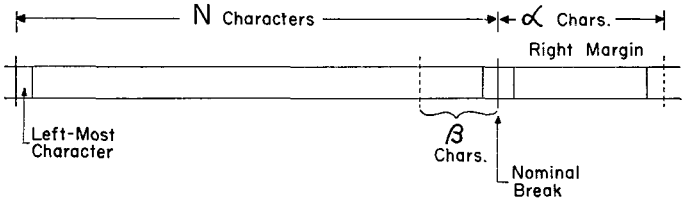

FIG. 1

Character Sets

The following character sets are considered by the routine.

(1) *Alphanumeric characters.* letters, digits, apostrophy (4-8 punch), plus zero (12-0 punch), minus zero (11-0 punch) and record mark (0-2-8 punch).

(2) *Vowels.* A, E, I, O, U, Y.

(3) *Special pairs.* SH, GH, PH, CH, TH, WH, GR, PR, CR, TR, WR, BR, FR, DR, vowel +R, vowel +N, OM.

(4) *Special characters.* blank, virgule, hyphen (same as minus), plus, asterisk, equals, right paren, comma, period, left paren and dollar sign.

(5) *Nonprinting characters.* The 5-8, 6-8 and 7-8 punches with or without an overpunch.

Method

A. First an attempt is made to find a *natural break.* This is accomplished by any of the following rules, whichever occurs first.

(1) If there is a blank or nonprinting character right after the nominal break, the line is retained as is.

(2) If a blank, nonprinting character, virgule or hyphen lies within β characters of the right margin, then the line is broken immediately following that blank, nonprinting character, virgule or hyphen.

(3) If a plus, asterisk, equals, right paren or comma lies within β characters of the right margin, and if a character *other* than a comma, right paren or period occurs to its right not later than the first character of the right margin, then the line is broken immediately preceding such noncomma, right paren or period.

B. If none of the above rules can be applied, then there is no natural break. In this case the rules below are

(*Please continue on the next page*)