Algorithm for the Optimization of Arithmetic Expressions in 1401 FORTRAN

Two inefficiencies which exist currently in the strings generated for arith- ~~aretu-~~ met~~r~~ic expressions by the 1401 FORTRAN compiler are discussed.

1. Redundant parenthesis generate redundant object time processing.
2. Generalization of the treatment of functions has led to inefficient output strings in specific cases.

The expression

```
A = (B) + (C)
```

generates

```
GT1 = B
GT2 = C
A = GT1 + GT2
```

The expression

```
A = B + C * D
```

generates

```
A = C * D + B
```

which is correct, but because

```
A = SINF (B) + C * D
```

would generate

```
A = SINF (C * D + B)
```

a rule was established which states that all functions force a generated temporary. Consequently, the above expression is generated as

```
GT1 = SINF (B)
A = C * D + GT1
```

which is correct.

But this rule leads to an inefficiency in the case of

A = SINF (B)

which produces

GT1 = SINF (B)

A = GT1

To correct this situation, the following algorithm is recommended.

Algorithm

Rule 1. If the operand immediately following the equal sign of a string is a generated temporary, the computation of the GT can be substituted for the operand and the GT string can be deleted.

Rule 2. When a GT occurs to the right of the equal sign, but is not the first operand, and all preceding operators have the same hierarchy, then by the rule of commutivity, the GT can become the first operand and procedure 1 will apply.

Exception 1. If the operator preceding the operand is "-", the negate function must be used.

A = B - GT1

is equivalent to

A = NEGATF (GT1) + B

Exception 2. If the operator preceding the operand is "/", optimization should not take place. The use of the invert function is unacceptable in fixed point computation and processing time might be increased in the case of floating point.

Exception 3. If any function computation precedes the GT, optimization cannot occur.

Example of Rule 1

A = (B * C) ** D

currently produces

GT1 = B * C

A = EXPF (LOGF (GT1) * D)

or in string notation

GT1 = BBB * CCC ‡ AAA = GT1$\underline{L}$ * D$\underline{E}$ $\underline{‡}$

By applying rule 1 we get

A = EXPF (LOGF (B * C) * D)

or in string notation

AAA = BBB * CCC $\underline{L}$ * D$\underline{E}$ $\underline{‡}$

Example of Rule 2

A = B * (C + D)

currently produces

GT1 = C + D

A = B * GT1

but by rule 2 it can be written as

GT1 = C + D

A = GT1 * B

and by rule 1 can be reduced to

A = C + D * B

where each operation is done serially (hierarchy does not apply).

The algorithm can be refined if it is conveniently programmable when more than one GT occurs in the string. Consider the following expression

A = (B + C) * (D * E) * (F * G)

This currently produces

GT1 = B + C

GT2 = D * E

GT3 = F * G

A = GT1 * GT2 * GT3

by applying rule 1, it can be reduced to

GT2 = D * E

GT3 = F * G

A = B + C * GT2 * GT3

No further optimization can occur. However, if the original expression had been written as

A = (D * E) * (F * G) * (B + C)

the string could have been reduced to

GT3 = B + C

A = D * E * F * G * GT3

It appears, therefore, that a third rule should be established which states that when an expression contains more than one GT, analysis of the expressions represented by the GT's should occur before optimization takes place. This rule will be harder to implement than the first two rules.

GM:meb

Gary Mokotoff
GP Advanced Programming Development