FORTRAN における最適化と問題点*

山　下　真一郎**

1. まえがき

FACOM 230—50 および 60 の FORTRAN に適用した最適化に基づいて，それによって発生した問題などについて述べる．

我々のコンパイラが，最高の出来栄えであると信じているわけではないが，なんらかの参考になれば幸いである．

2. 最適化の意味

最適化とは，目的に最も良く適合させるということで，目的をはっきりさせることが重要である．最適化はその目的を実現するとき，同じ効果がより少ない費用で実現するか，または，同じ費用で，より多い効果を実現することにある．

いずれにしても，最適化はその目的を実現する手段を適正に選ぶことにある．しかも，できるだけ多くの手段の中から適正なものを選ばねばならない．また，結果に及ぼす影響の大きい手段の順に最適化するのが効果的である．結果に及ぼす影響が少ない手段をどんなに最適化しても，効果は少ない．

我々が計算機を使うのは，それが目的ではない．ましてや，FORTRAN を使って，計算機を働らかすのは，手段のほんの一部であろう．計算機を使うべきかどうか，FORTRAN を使うべきかどうかを最初に検討すべきである．

FORTRAN を使うことが決定したとして，同じ結果を得る FORTRAN の記述法は無数にある．記述法は計算法とその記述を適正に選ぶことが重要で，これを第2に検討すべきであろう．

これから議論する最適化は，FORTRAN で，計算法も記述も固定されたのち，これを機械語に変換した目的語の効率に関するものである．それは，計算機時間——すなわち費用——を少なくすることを目標にする．これは副次的に計算機の記憶容量も少なくすることが多い．

3. 目的語を最適化することの是非

最適化の目標である“計算機時間の短縮”は，いろいろな手段の中から，たとえば，

（1） 計算機を使う．
（2） FORTRAN を使う．
（3） 計算法を決める．
（4） 記述を決める．

の順に手段を決定するのであるから，この順に最適に決定するのが効果が大である．すなわち，処理系がどんなに最適な目的語を作るように努めても，言語や計算法，記述を最適に選ぶ方がより“計算機時間の短縮”のために役立つ．すなわち，目的語を最適化する処理系を選ぶことは下の下の手段であり，さらに，そのような処理系を作る必要がないという次のような意見もあり，また，それぞれに反論もある．

3.1 最適化不要論

（1） 計算機はしだいに高速化し，最適化コンパイラを作る労力を計算機自身を高速化することに投入した方がよい．

（2） 最適化コンパイラは翻訳時間がかかるから，短縮した目的語の実行時間よりも，翻訳時間の方が大きくなる．

（3） 最適化コンパイラの目的語は，原始プログラムと違ってくるから，デバッギングが困難である．

（4） 言語の規則を最大限に活用するために，規則の適用が厳格になって，プログラムが作りにくい．

（5） 最適化の大部分はプログラマが行なえる種類のものであるから高価な計算機を使うよりも，安価なプログラマが最適化すべきである．

（6） 最適化のもう一つの部分は，計算機に依存する部分であるが，言語に適するような計算機を作ればこの部分は不要になる．

3.2 最適化必要論

（1） 現在の使用可能な計算機の限界近くで動作するプログラムが存在して，最適化コンパイラを作って

---

* Optimization and Related Problems on FORTRAN, by Shinichiro Yamashita (Fujitsu Limited)

** 富士通株式会社

も十分採算が合う.

(2) 最適化項目をオプションにして，プログラマに選択させればよい.

(3) プログラマがデバッギングの方法を考えることが不可能ではない.

(4) 言語の規則はもともと厳格であるべきで，規則を拡大解釈すべきではない.

(5) すべてのプログラマが，自分のプログラムを最適化できるほど十分な知識を持っているわけではないし，ある時には，入念に最適化するだけの十分な時間的余裕がないことがある.

(6) 万能な計算機を作るためには，特定の言語に依存するようにするわけにいかないし，金物では計算機ごとに費用がかかるが，コンパイラは1回だけしか費用がかからないから十分安くなる.

最適化コンパイラを作ることには，以上のように賛否両論があって，それぞれの立場に立てば反論の余地がない．したがって，コンパイラを一つしか作らない場合には，最適化の程度が問題である．すなわち，そのコンパイラ経由で持込まれる問題の翻訳時間と実行時間の和が最小になるような程度に最適化すべきである．また，プログラマが最適化できるところはプログラマが最適化すべきで，処理系はプログラマが手出しできないところを重点的に最適化すべきである.

4. 最適化の種類

最適化には，言語に依存するものと計算機に依存するものがある．ここでは，言語については FORTRAN を中心に述べるが，他の言語でも同様の議論が展開できよう．計算機はある程度の規模の計算機ならば備えている機能を想定する.

最適化は，言語と計算機に依存するけれども，次のような項目について考慮することになる.

4.1 最適化項目

(1) 共通式の消去

同じ計算結果をもたらす式は1カ所で計算して残りの式を消去する.

(2) 定数の計算

定数および定数を暗に含む計算は翻訳時に計算する.

(3) 配置転換

頻度の高い計算で，そこで計算する必要のないものは，計算頻度の低い所で計算する.

(4) 命令の置換

計算時間のかかる命令は速い命令で置きかえる.

これらの場合の例題と問題点を説明する．最適化はプログラム全体で考慮することが望ましいが，翻訳時間を考慮すると，ブロックに分割して考慮するのがよい．ブロック化は次の点を始点として行なう.

4.2 ブロッキングの目安

(1) プログラムの最初の実行文

(2) ENTRY 文の次の最初の実行文

(3) ブロックの終りの次の最初の実行文

(4) DO 文

(5) 参照される文番号を持つ実行文

(6) CALL 文

(7) 端末文の次の最初の実行文

ブロックは上記の点を始点として，次のブロックの始点の直前または END 行の直前までの連続する実行可能文の集りである．要するに，ブロックは最初が入口であるようなプログラム部分で，これを満足すれば，任意の所で分割してもよい．したがって，一つのブロックがあまり大きくならないように分割した方がよいときは，上記の点で分割を行なったのちに，他の任意の点で分割すればよい.

5. 共通式の消去

原始プログラムは適当な中間語に変換して翻訳を進めるが，特に数式は3アドレスの表現にしておく．たとえば，それを $A=B+C$ とする．共通式の消去は，ブロック中で行なう．このため，ブロック内の中間語は計算の流れの順に，次に述べるような方法で TABLE に stack する.

この TABLE 作りは，消去の調査と同時に行なう．$B+C$ が消去可能かどうかを調べるとき，TABLE を計算の流れと逆向きの方向に調べる.

その調査は，まず，$B+C$ が消去できない場合であるところの,

(1) B が定義されている.

(2) C が定義されている.

(3) ブロックの始めまで調べた.

を確認し，これにあてはまれば，$A=B+C$ を stack する．これにあてはまらなければ，$B+C$ が消去できる場合であるところの,

(1) $B+C$ と同じ表現である.

(2) $C+B$ と同じ表現である.

を調べる．これらがすべてあてはまらなければ，次の TABLE を同じ順に調べることになる．もし，消去で

きることが確認できれば，その発見されたところの結果で，現在の $B+C$ を置き換える．

消去可能性を調べるとき，置換可能な演算である加算，乗算では，置換して調べる．

この最適化は単純であるが，次のような問題点がある．

(1) 消去することが optional になっている処理系では，消去したときとしないときの計算結果が相違することがある．

たとえば，次の例では，Y の値が異なるかも知れない．

$$X=(B+C)$$
$$\vdots$$
$$Y=(B+C)+D$$

データが一般の記憶装置にあるよりも，演算レジスタにあるときが情報精度が高いとすれば——演算レジスタ内において，単精度と倍精度の区別をしない処理系では，しばしば，このような関係にある——($B+C$) を消去しないときは，Y の評価で，($B+C$) の値は演算レジスタに置かれ，それに D が加えられるだろう．($B+C$) を消去するときは，($B+C$) は記憶装置内にある可能性が高い．このような事情で，Y の値が異なって，増幅されると，大きな違いとなりうる．

このような現象は，積極的に解釈するとすれば，数値計算的な安定性を確認する手段になることを注意しておきたい．

(2) 関数はその効果を考えると，是非消去したい項目であるが，JIS では，一つの文内でのみ許されている．

直接または関接に入出力命令を含む関数や値を遷移するような関数——たとえば，乱数の関数——を参照するとき問題が起きる．

FORTRAN では，一つの文の式の評価の順序が規定されていないから，上記のような関数は一つの文に同じ関数を2回書くと結果がどのようになるか不明である．さらに，同じ装置に対する入出力命令を含む異なった関数を同一の文に書いても，その結果がどのようになるか不明である．

これらの点があまり釈然としないから，FORTRAN では，同じ引数に対しては，いつでも同じ結果をもたらすものしか関数とすべきではなく，同じ引数に対して，いつでも同じ結果をもたらすとは限らないものは，サブルーチンとすべきではないかと思う．

(3) 演算の組合せ変更の問題

同じ強さの演算子が続くときなどに考えられる問題である，たとえば，$X=A+B+C$ に，$A+B$, $B+C$, または $A+C$ の計算を先きに実行することが考えられる．この方法の利得は，コンパイラの負担のわりには少ないように思う．また，プログラマが意図しないトラブルも発生することが考えられる．

6. 定数の計算

定数と定数の計算を翻訳中に実行することは，あまりむずかしくない．

定数を暗に含む計算を翻訳時に実行するには，共通式の消去と同じ方法で実現することができる．

たとえば，半径 R の球の体積 V を求めるとき，

```
PI=3.14159
V=4.0*PI/3.0*R**3
```

と書くことは多い．これらの中間語は，次のように表わせよう．

(a) PI=3.14159
(b) T_1=4.0*PI……=12.56636
(c) T_2=T_1/3.0……=4.18879
(d) T_3=R**3
(e) V=T_2*T_3

共通式の消去と同じ方法で TABLE を作る．4.0*PI を調べるとき，PI の定義があるかどうか，あればその右辺は定数かどうかを調べる．今の場合は，(a) のところで，PI が定義されていて，右辺が定数であるから，4.0*PI は計算できて，T_1 は定数となる．同様に，T_1/3.0 は計算できて，T_2 も定数となる．結局これらの式は，次の式を計算することになる．

```
PI=3.14159
V=4.18879*R**3
```

もし，PI がプログラムのどこでも使われていないならば，この代入文と定数 3.14159 を消去できるはずであるが，ここで述べた方法だけでは困難である．

この最適化には，次のような問題点がある．

(1) 同じ強さの演算子が続くとき，入れ換えを行なうかどうか，たとえば，4.0*A/3.0 を (4.0/3.0)*A とするかどうかの問題である．このときはオバーフローやアンダーフローなどが起きる可能がある．

7. 配置転換

計算頻度の高い一部のプログラムを計算頻度の低い所で計算しておいても，得られる結果が同じならば，

そのように配置転換を行なうことは非常に好ましい.

たとえば，次のようなプログラムを考える.

```
      DO 10 I=1, N
10    X(I) = (A+B)
```

(A+B) は DO の範囲内で計算する必要がない．したがって，このプログラムを次のように変形できるとよい.

```
      T1 = (A+B)
      DO 10 I=1, N
10    X(I)=T1
```

このような例は，loopを形成する個所で考えることができるが，DO loop のみに適用する方が実現は容易である.

実現の方法は，DO ブロック内で定義されている変数について，DO ブロックごとに TABLE を作り，中間語の各変数がこの TABLE になければ，この中間語は，その DO ブロックのそとへ出せる．このような操作は，共通式の消去を行なう前に行なう.

このプログラムの再配置の問題は，いろいろな手段によって行なうことができるが，あまりに，簡便な手段によって行なうと，次のような問題が発生する.

(1) 改善されるよりも，むしろ悪化される場合がありうる．これは，正確に計算の流れを把握していないために発生するものである．たとえば，次のような例を示すことができよう.

```
      X=1.0
      DO 1 I=1, N
      IF(X) 2, 1, 1
2     Y=(A+B)
1     CONTINUE
```

このプログラムは Y=(A+B) を計算しないが，(A+B) が DO に関係がないので，DO のそとで計算する．しかし，明らかに，DO のそとで計算すれば損である.

(2) 誤りを引き起す場合があり得る．これも(1)と同じような場合で，AまたはBが定義されていないとき，(A+B) を計算することによって，オバーフローなどのような困難な事態が発生するかも知れない．次の例ではもっとめんどうなことになる.

```
      DIMENSION A(10)
      DO 1 I=1, 10
      DO 2 J=1, 5
      IF(I .GE. 2*J) GO TO 1
      X=A(I+1)-A(I)
2     CONTINUE
1     CONTINUE
```

上記のようなプログラムは以下のように変形されよう.

```
      DIMENSION A(10)
      DO 1 I=1, 10
      T1=A(I+1)-A(I)
      DO 2 J=1, 5
      IF(I .GE. 2*J) GO TO 1
      X=T1
2     CONTINUE
1     CONTINUE
```

この結果，I=10 のとき，A(I+1) は宣言の範囲を越えてしまう．これはプログラムの領域を越えて，重大な誤りとなるかも知れない.

同様に，次のような例題では，

```
      DO 1 I=1, N
      A(I)=0.0
      IF(X .GT. 0.0) A(I)=SQRT(X)
1     CONTINUE
```

次のように変形すると，$X<0$ のとき，SQRT(X) がエラーを引き起す.

```
      T1=SQRT(X)
      DO 1 I=1, N
      A(I)=0.0
      IF(X .GT. 0.0) A(I)=T1
1     CONTINUE
```

これは次のように翻訳できれば，最適化されることになろう.

```
      T1=0.0
      IF(X .GT. 0.0) T1=SQRT(X)
      DO 1 I=1, N
      A(I)=T1
1     CONTINUE
```

しかし，人手で行なうほどに簡単に，このように翻訳することは困難であろう.

(3) DO の拡張範囲があるときは，次のような問題が起きる.

(a) 拡張範囲で定義される変数があるかも知れないために，DO のそとで計算してよいかどうかの判定が困難である.

(b) 式の中間結果の記憶装置や演算レジスタなどの利用法が重複しないようにしなければならない.

これは，最適化を行なわない場合にも考慮すべき点

で，記憶装置の節約のつもりがかえって節約にならない場合が多く，拡張範囲を認めることがよいことかどうか疑問である．

8. 命令の置換

計算時間のかかる命令を計算時間のかからない命令で置き換える問題は，最適化の最大の関心事である．

べき乗算は普通には，サブルーチンで計算されるから，整定数べき乗のものは乗算に置き換えるべきである．

乗除算において，2進法の計算機では，2のべき乗の定数を含めば，シフト命令または指数部の加減算命令を用いる方法もある．乗算を加算で置き換える方法はアドレス計算で最も有効である．

加減算において，回帰的に定義されている式，たとえば，$I=I+C$ は I をレジスタに割り当てることができれば，Load/Store 命令を節約することができる．また，C が適当な定数ならば，Immediate 命令の加算で行なうことができる．これらは特に，DO の制御変数に対して有効である．

命令の置換ではないが，論理式の評価において，一部しか評価しない方法が考えられる．たとえば，OR でつながれた次の式を二つの式に分ける方法である．

```
IF (A·EQ·B·OR·C·NE·D) GOTO 10
                ↓
{ IF (A·EQ·B) GOTO 10
  IF (C·NE·D) GOTO 10
```

アドレス計算について，もう少し詳しく述べよう．配列の宣言を $A(d_1, d_2, \cdots\cdots, d_n)$ とし，配列要素 AE を $A(p_1*I_1+q_1,\ p_2*I_2+q_2,\ p_n*I_n+q_n)$ とする．d_0 を A の1語の大きさ，$A_0=A(1, 1, \cdots\cdots, 1)$ とすれば，AE は次のようになる．

$$AE=C_0+\sum_{j=1}^{n} C_j*I_j$$

ここに，
$$C_0=A_0+\sum_{j=1}^{n} D_{j-1}*(q_j-1)$$
$$C_j=D_{j-1}*p_j$$
$$D_j=\prod_{k=0}^{j} d_k$$

最適化の対象になるのは C_j*I_j の扱いである．I_j が DO の制御変数であれば，この乗算を加算で置換すると同時に，前に述べた共通式の消去や配置転換などと組み合わせて効果を上げることができる．次の例題で説明しよう．

```
DO n1 I1=m11, m12, m13
DO n2 I2=m21, m22, m23
 ⋮
DO nk Ik=mk1, mk2, mk3
 ⋮
DO nn In=mn1, mn2, mn3
┌──────────────────────────────────┐
│ A(p1*I1+q1, p2*I2+q2,…, pn*In+qn) │
└──────────────────────────────────┘
nn  CONTINUE
 ⋮   ⋮
n1  CONTINUE
```

このプログラムは，次のように変形される．

```
        I1=m11
        J1=C1*m11
  L1
        I2=m21
        J2=C2*m21+J1
  L2
        ⋮
        Ik=mk1
        Jk=Ck*mk1+Jk-1
  Lk
        ⋮
        In=mn1
        Jn=Cn*mn1+Jn-1
  Ln
        ┌──────────┐
        │ AE=C0+Jn │
        └──────────┘
        Jn=Jn+Cn*mn3
        In=In+mn3
        IF (In·LE·mn2) GO TO Ln
        ⋮
        Jk=Jk+Ck*mk3
        Ik=Ik+mk3
        IF (Ik·LE·mk2) GO TO Lk
        ⋮
        J2=J2+C2*m23
        I2=I2+m23
        IF (I2·LE·m22) GO TO L2
        J1=J1+C1*m13
        I1=I1+m13
        IF (I1·LE·m12) GO TO L1
```

このような変形は基本的なもので，これ以上の変形は最適化の程度によって異なる．I_k, J_k をそれぞれレジスタに割り当てる方式や，C_k が1のとき，I_k と J_k を同一レジスタに割り当てる方式や I_k が他の目的に使われていないとき，I_k と J_k を同一レジスタに割

り当てて，判定を変形するなどが考えられる．いろいろな大きさの配列が使われているときは，レジスタをじょうずに割り当てるかどうかで大きな違いが生ずる．レジスタがどうしても不足する場合には DO の nest の深い方の，参照頻度の高い方からレジスタに割り当てて，残りは記憶装置に割り当てることになる．

この最適化は言語に依存するよりも計算機に依存し，あまり問題が起きない．しかも，プログラマが関与できない部分の最適化であるので，最初に，そして，最もよく考慮されねばならない最適化である．この最適化に何も問題がないとはいえ，次のような問題点がある．

（1） べき乗を乗算で置き換える場合，大きなべき指数でも，乗算で展開してよいかどうか，あるいは，大きいべき指数は一律に規制すべきではないか．

（2） DO が満足されたときの DO の制御変数は不定になる約束になっている．これは，DO loop の途中からは抜け出さないプログラムならば，制御変数の代わりをレジスタに割り当てて，記憶装置の制御変数は値を保持しないようにするためであり，また，制御変数が DO loop 内で有効に使われていないならば，アドレス計算用のレジスタで代用して制御変数を消去するためである．しかし，DO が満足されたとき DO の制御変数が特定の値を持つことを期待しているプログラマは多い．

（3） 添字式は制限された形式しか許されていないが，これが拡張されることを望むプログラマは多い．この要望は少ない手間で最適化を行なうことをさまたげる．

9. あとがき

最適化問題はこの他に言語仕様と対立するものとして，組み込み関数の扱いや FORMAT の分解問題などがある．また，実行時の誤りの検出との問題もある。誤りは，往々にして誤りが起きるかも知れないとおそれているときは起きず，誤りが起きるはずがないと信じているときに起きるため，optional な誤りの検出法では手遅れになる恐れがあることが問題である．

最適化は各部分のバランスをよく保つ必要があるから，正確な統計に基づいて推進する必要がある．そして，一つの計算機システムで何個かのコンパイラの開発が必要であろう．それは，コンパイラ自身と目的語を統計に基づいて平均に改善しても，要求の全てを満足できないからである．

コンパイラを作り，実用に供する立場からは，最適化を強力に進めると翻訳時間がかかるのもさることながらコンパイラが複雑になって，安定に動作するまでにかなりの時間を要することも問題である．

最適化評価はいろいろな場合が考えられるので，一つの事例で判断するのは意味のないことであるが，我々の得た結果を示しておく．

コンパイラAは上から下へ何も細工せずに翻訳する．コンパイラBは命令の置換，おもにレジスタの最適化を行なう．コンパイラCは命令の置換や命令の転置など，ここに述べた最適化を行なうものである．20元連立一次方程式を消去法で解くプログラムにおいて，時間比を調べると，翻訳時間比は A：B：C＝45：87：100，実行時間比は A：B：C＝328：103：100 である．

連立一次方程式は3重の loop があり，loop の中の命令を loop のそとへ移動できるものはほんのわずかで，アドレス計算の最適化が大きな比重を占めている。

コンパイラA，B，C（特にB）はそれぞれ最適に作ったわけではないので，翻訳時間比の意味づけはあまり役にたたない．

結論的には，大多数のプログラムの翻訳，実行時間を短縮するためには，中間語の参照頻度を多くしないような程度の最適化と言語仕様を選定し，最適化の程度を選択できるようにしない方がよいのではないかと思われる．そして，余裕があれば，長大なプログラムに対処するために，徹底した最適化とそれにふさわしい言語仕様を決めるべきだと思う．

参考文献

1) Edward S. Lowry and C. W. Medlock: Object Code Optimization, Comm. ACM **12**, 1 pp. 13～22 (Jan. 1969)
2) Vincent A. Busam and Donald E. Englund: Optimization of Expressions in FORTRAN, Comm. ACM **12**, 12 pp. 666～674 (Dec. 1969)
3) F. F. Allen: Program Optimization, Annual Review in Automatic Programming pp. 239～307 (1969)
4) C. W. Gear: High Speed Compilation of Efficient Object Code, Comm. ACM 8, 8 pp. 483～488 (Aug. 1965)

（昭和45年4月23日受付）