5. コンパイラ

大駒誠一，近藤頌子，土居範久，原田賢一
（慶応義塾大学工学部管理工学科）

1. まえがき

コンパイラを作るといつても算術式の翻訳さえできればほとんどできあがりというような簡単なものから，数多くの入出力装置をとりあつかい，文法に対する制限もなく，他のいくつかのシステムと関連をもつた大型のコンパイラまでいろいろあつて一概に言えないが，その主な作成上のプログラムテクニックは後入れ先出しの原則をもつ棚（stack）とリスト構造の使用だろう．後はサブルーチンを上手に作つて，適当に組合せればよい．

ここではソースプログラム言語にはFORTRAN，ALGOL，COBOL のみをとりあげる．最近 NPL が発表されたが，これの作成にはさきの3つのコンパイル上のテクニックを使えば十分と思われるので省略した．

2. FORTRAN， ALGOL，COBOLの比較

2.1 各言語の特徴

FORTRAN，ALGOL，COBOL は同じ問題向き言語とはいつても，その発生過程や使用目的などによつてそれぞれ特徴があり，その主なものを次にあげ，後は一覧表にした．

FORTRAN

- array以外の変数が宣言せずに使用できる．
- 複素数演算，2倍精度演算もできる．

ALGOL

- プログラムがblock構造をもつている．
- recursiveなprocedure callができる．
- procedureの名前による呼び出し．
- arrayのサイズが実行時にきまる．

COBOL

- プログラムを4部にわけて書く．
- データは文字単位に長さをきめる．
- レコードとかつファイルの概念にともなつてデータの移動ができる．
- データがTree構造をもつている．

第１表　　FORTRAN, ALGOL, COBOL の比較

	FORTRAN	ALGOL	COBOL
使用する文字	A～Z，0～9，＝＋－＊／（），。$	a～z，A～Z，0～9 ＋－×／÷↑<≦＝≧>≠≡ ⊃ ∨∧¬，．10：；（）［］“ ”	A～Z，0～9 ＋－＊／＝$，．；”（）<>
identifier	英数字の組合せ １字目は英字で始まる 長さは６字以内	英数字の組合せ １字目は英字で始まる 長さは任意	英数字の組合せ，“－”が中に入つてもよい 英字を１字含むこと 長さは３０字以内
label	５桁以内の符号なしの整数	同上および符号なしの整数	同上および符号なしの整数
データのタイプの定義	１桁目できめ宣言せず （FORTRAN Ⅳでは宣言もある）	使う前に宣言する	DATA DIVISIONで定義
label の定義	ステートメント番号欄（1～5col.）に書く.	labelと“：”	Col. 8から始まるlabelと“．”とそれに続くスペース
データの単位	語単位	語単位	文字単位
データの構造	COMMON 以外はバラバラで独立	block 内でそれぞれ独立	77以外は tree構造
データ	整数，浮動小数点数，Boolean，２倍精度数，複素数，Hollerith data	整数，浮動小数点数，Boolean，（string）	整数，小数点のある数，．英数字のitem，group item，レコード，ファイル
array	最高３次元 大きさ固定 （FORTRAN IVはadjustable dimensionあり）	次元数任意 大きさはdynamicにきまる	最高３次元 大きさは通常固定

	FORTRAN	ALGOL	COBOL
subscript	C, V, V±C, C＊V C＊V±C′	任意の数式	整数と整数の変数
wordの切れ目	＊, ／, ＝, (,), など DO2I=J,K REAL A,B,C } など例外あり	＋, －, ×, [,], goto, for, until など delimiter	スペース, , , ; , . とそれに続くスペース
statement の切れ目	次にcontinuationのないカードの終り	;	;, THEN, ., とそれに続くスペース, 次の動詞
statement のタイプの決定	全体を見渡す	最初の delimiter	最初の key word
算術式	simple arithmetic expressionのみ mixed modeは不可	Boolean, designational expressionも可	simple arithmetic expressionのみ
組み込みサブルーチン	SIN, SQRT, EXP, TAN, DIM, MAX など	cos, abs, sqrt, ln, exp, entier など	な し
function typeサブルーチン	arithmetic statement function. FUNCTION subprogram	type procedure	な し
procedure typeのサブルーチン	SUBROUTINE subprogram	procedure	DEFINE
サブルーチンの呼び出し	function name CALL subroutine name	procedure name	new verb name PERFORM INCLUDE
remarks	後から文法が整備されたので作成上の問題点は少い.	文法は非常にスッキリしているがすべてを完全に実現するのはむずかしい.	読みやすさに重点をおいているため冗長度が多い.

2.2　各言語の問題点

2.2.1　FORTRAN

A)　delimiterの不完全さ

DOとかREALなどが必ずしもdelimiterとなるとはかぎらないために，ステートメントのタイプがステートメント全体を見渡さないと決定できない（右側は同じ意味のALGOLプログラム）.

```
REALA,B,C          real  A,B,C
REALA=B+C          REALA:=B+C
DO2I=J,K           for  I:=J  step 1 until K do
DO2I=J+K           DO2I:=J+K
```

B)　EQUIVALENCE

```
DIMENSION      A(2), B(2), C(3), D(2)
COMMON         A, B, C
EQUIVALENCE    (C(3), D(1)), (B, E)
```

上のようなspecification statementがあるとき，その番地の割りつけは次の左のようになる．もしEQUIVALECEがなければ右のようになる（1000語の計算機とした）.

FORTRANⅡ

999	C(1)		A(1)
998	C(2)		A(2)
997	C(3)	D(1)	B(1)
996		D(2)	B(2)
995	B(1)	E	C(1)
994	B(2)		C(2)
993	A(1)		C(3)
992	A(2)		

EQUIVALENCEがある場合　　EQUIVALENCEがない場合

EQUIVALENCEがCOMMONやDIMENSION に関連して使われると番地割りつけは非常に複雑になる．しかもEQUIVALENCE はプログラムのどこに来てもかまわないので，ソースプログラムをいつたん全部読んでみるまでは番地の割りつけをすることができない.

2.2.2　ALGOL

A)　procedureのrecursive call

```
real procedure SUM(n); integer n; array a[1:n];
    if n=0 then SUM:=0 else
    SUM:=a[n]+SUM(n−1)
```

このようなprocedureのrecursive callについては実現が難しいわりに使用することはまれでこういった自分自身を呼ぶprocedureの呼び出しは禁止しているALGOLコンパイラは非常に多い．実際にこの例も次の様にするのが普通で効率も良い．

```
real procedure SUM(n);
value n; integer n; array a[1:n];
   begin integer i; real s;
      s:=0;
      for i:=1 step 1 untin n do s:=s+a[i];
      SUM:=s
   end
```

B)　procedureの名前による呼び出し

```
begin integer i; real array a[1:n], b[1:n]; real s;
   procedure summ(s, f, n); real s;
      begin s:=0;
         for i:=1 step 1 until n do s:=s+f
      end;
   summ (S, a[i], 100);          ..... (1)
   summ (S, b[i×2], 100);        ..... (2)
   summ (S, 1/i−ln(i), 100);     ..... (3)
end
```

のプログラムで(1), (2), (3)で呼んだ結果は次のようになる．

(1)　$S=a[1]+a[2]+\cdots\cdots+a[100]$

(2)　$S=b[2]+b[4]+\cdots\cdots+b[200]$

(3)　$S=1+\frac{1}{2}+\frac{1}{3}+\cdots\cdots+\frac{1}{100}-(\log 1+\log 2+\cdots\cdots+\log 100)$

C)　Type

procedureのformal parameterでtype の宣言してないものはとりあつかいには注意を要する．

```
begin real a, b, c, d; integer i, j;
   procedure P(x, y, z, w);
      begin z:=x+y;
            w:=x÷y
      end;
   P(a, b, c, d);          ..... (1)
```

P(i, j, c, d) ..... (2)

end

この例で(1)で呼ぶと c には正しく a と b のrealの和が入るが d は未定義となる．(2)で呼べば両方ともvalidになる．このようにprocedureのbody は何通りものコンパイルをするか，パラメータのタイプを実行時に判断して演算を変えなければならないことがある．

2.2.3 COBOL

A) word separatorがあつてもなくてもかまわないため全く同じものに何通りかの書き方ができる．

```
MOVE A TO B C
MOVE A TO B, C
MOVE A TO B, AND C
```

はすべて同じ．

```
MOVE A TO B ,AND C
```

は誤り．コンマの前はブランクではいけないし，後はブランクでなければいけない．
statement separator (; とTHEN)についても同様．

B) 中間結果

```
COMPUTE A= B*C*D/(E*F)
```

なる式をコンパイルする際に，文法通り有効数字を失なわないように中間結果は十分な桁数をもつていなければならない(COBOL－61 EXTENDED) とすると，変数がそれぞれ10数桁ある場合には中間結果の桁数は非常に厖大になる．実際の事務計算ではこのようなことはほとんどないが．

C) 複数個のタイプのレコードをもつファイル．

```
DATA DIVISION.
FILE SECTION.
FD A－FILE LABEL RECORD IS OMITTED
   DATA RECORDS ARE  ABLE, BAKER, CHARLY.
01 ABLE.
   02 CARD－NO   SIZE IS 1
      88 TYPE－1   VALUE IS 1.
   02
      .
      .
      .
```

```
01  BAKER.
    02  CARD-NO    SIZE IS 1
        88  TYPE-2    VALUE IS 3.
    02
        :
        :
        :
01  CHARLY.
    02  CARD-NO    SIZE IS 1
        88  TYPE-3    VALUE IS 3.
    02
        :
        :
        :
```

上記のようなデータの指定のあるとき

```
READ A-FILE; AT END GO TO SOMEWHERE.
```

ではA－FILEのどのレコードを読んだのか解らない．実際に使う前に

```
IF TYPE-1 THEN MOVE ABLE TO WORK-1 OTHERWISE
IF TYPE-2 THEN MOVE BAKER TO WORK-2 OTHERWISE
MOVE CHARLY TO WORK-3.
```

とすればよいが，condition nameを使わずに，

```
IF CARD-NO IN ABLE EQUALS "3" THEN MOVE·····
```

としてまだ解らないrecord－nameで修飾するのは嫌な気持である．

D)　pictureの終りのピリオド

DATA DIVISIONで

```
05  ALPHA  PICTURE IS 99.
05  BETA   PICTURE IS 99..
05  GAMMA  PICTURE IS 99. SYNCHRONIZED RIGHT.
```

なるデータの指定では実際のpictureはそれぞれ

```
ALPHA    99     (２ケタ)
BETA     99.    (３ケタ)
GAMMA    99.    (３ケタで１word)
```

である(numeric literalで99.は誤りである)．

E)　SIZE ERROR

例えば

```
    MULTIPLY A BY B, C, D AND E ON SIZE ERROR
    GO TO SIZE-OVER.
```

でSIZE−OVERにコントロールが移つたときにはB, C, D, E のうちどれがオーバーフローしたかわからないし，またそのうちのいくつがオーバーフローしたかもわからない．

F） IFにおける省略形

```
    IF A=10, B, C OR D
```

は

```
    IF A=10 OR A=B OR A=C OR A=D
```

のことである．また，

```
    IF A EXCEEDS B OR EQUALS B AND X EQUALS Y
    THEN IF GREATER THAN B MOVE C TO B.
```

は

```
    IF A EXCEEDS B OR (A EQUALS B AND X EQUALS Y)
    THEN IF X IS GREATER THAN B MOVE C TO D.
```

G） COPYと修飾

DATA DIVISIONで

```
    01 A
       02 B
          03 C
          03 D
       02 E
       02 F COPY G
       02 G
          03 C
             04 H
             04 I
       02
```

なるデータの指定があるとき

```
    H IN C IN G IN A
    H IN C IN G
    H IN G
```

はいずれも同じものをさして正しく，

```
    H IN C
    H IN A
```

はユニークでないので誤りである．修飾はユニークになりさえすれば適当に省略してかまわないので，コンパイラは大変である．

H）　fixed portion

SEGMENT LIMTから49までの優先番号をもつSECTION は番号の小さい順に記憶装置に入るだけ fixed portion （常時記憶装置にある）に入れるが，それがわかるのはコンパイルのほとんど最後の段階で機械語になるときにきまるので，オペランドが fixed portion に入つているか否かによつてコンパイルの違うものについては一番最後までオブジエクトプログラムは確定しない．

3．コンパイルの方法

普通コンパイルするにはソースプログラムを読み込みながら文法的な誤りを検査して，適当なオブジエクトプログラムを作り出すのであるが，その方法はそのプログラムの使い方，下に持つている中間言語の性能，計算機の大きさや構成などによつて違うし，無論言語の相違によつても異なつてくる．

一般にコンパイル時に手間をかければ，効率のよいオブジエクトプログラムが出てくるがコンパイルに時間がかかる．これと反対に，実行時のための各種のサブルーチンを多く用意しておいてコンパイラはそれにリンクをとらせ，いろいろな判断は実行時にやらせるようにすれば実行時間は遅くなるがコンパイルは簡単になる．

3.1.1　使い方によつて

ルーチンワークをするものについてはコンパイルに時間をかけていいオブジエクトプログラムを出すようにし，1回限りの仕事についてはコンパイルは手早くやるが実行時の効率はよくないといつた二通りのコンパイラを用意しておいて使い方によつて任意に選択できると良い．

またこれとはややニユアンスが違うが実行時の速度を早くするか，使用する命令数を少くするか選択できるようになつている場合もある．

3.1.2　中間言語によつて

コンパイラはソースプログラムを直接機械語には変換しないで，いつたん既製のアセンブラなどの中間言語に落し，それ以後の処理はアセンブラにまかせるといつたコンパイルのやり方は非常に多い．この場合アセンブラの性能によりコンパイルの方法は異なり，ただ機械語を記号で書くだけのような簡単なアセンブラでは，コンパイラの労力は直接機械語に落すのとアセンブラに落すのとではあまり変りがない．アセンブルするだけ翻訳時間が増えて損になる場合もある．しかしながら，FORTRAN プログラムの中に一部アセンブラで書いた

ものを入れたいとか，他の言語もすでにアセンブラを通しているときなどプログラムの連結をはかるためには中間言語を媒介とするのは通例で便利である．

アセンブラがもつと強力で，各種のマクロ命令やIOCSなど含み，しかも条件付翻訳ができるときにはコンパイラはアセンブラに落す方が簡単であるし効率も良い．特に条件付翻訳は良いオブジエクトプログラムを作り出すためには非常に有効でコンパイラの中間言語として是非ともそなえていたいものである．

3.1.3　計算機によつて

主記憶装置が十分あつてコンパイラが全部入り，しかもコンパイルされたオブジエクトプログラムも十分入る余裕があれば，中間結果を外部記憶装置に出すわずらをしさもなく，翻訳時間も短くて非常に好都合である．しかし，ルーチンワークものの場合は翻訳時間がよほど短くないかぎり，機械語に変換しておいた方が得のようである．

これとは逆に比較的小さな計算機で，紙テープやカードのように非常に遅い入出力装置しか持つていない場合，文法にいろいろな制限をつけて言語を簡単にし，中間結果を出さずに，ソースプログラムを全部読み終るとオブジエクトプログラムが記憶装置のコンパイラ以外の部分に完成するようにしておくとすぐそのまま実行に移れて手軽で使いよい．一般に文法にかなりの制限をつけても実用上それほどさしつかえのない場合が多いようである．

多くの場合磁気テープやデイスクに中間結果を書き出して何回かそれをやりとりしてオブジエクトプログラムを作り出すのであるが，そのパスはオブジエクトプログラムの効率とのかねあいであるが少いにこしたことはない．

一般にコンパイルに手をかければ良いオブジエクトプログラムができるし，コンパイルを簡単にするには，オブジエクトプログラムができるだけサブルーチンを使うようにすればよい．しかし小さい計算機ではサブルーチンだけで記憶装置がいつぱいになつてしまのこともあり，小さい計算機程手間をかけて良いプログラムを出さなければならない場合もある．

この他 push down memoryをもつている計算機では，算術式はpolish notationに直すだけでいいし，procedureのrecursive callも比較的簡単にでき，コンパイル時に頻繁に使うstack にも好都合である．この他リストの処理が簡単にできる命令とか table look atができるとかいつたことはコンパイラにとつて有利になる．

3.2　コンパイルのしかた

算術式やブロック構造の処理などコンパイラの中枢となるべき部分のテクニックについていくつかの文献から抜萃して紹介する．COBOLの作り方については[12]に詳しい．

3.2.1　算術式（Arithmetic expression）

コンパイラの主格的存在たる算術式の翻訳処理方法として，今日種々様々な方法が考案されている．これらの処理方法は

1°　逐次処理方式（Sequential translation）

2°　非逐次処理方式（Non−Sequential translation）

の二種に大別できる．

逐次処理方式とは，読み込みながら，オブジエクト・プログラムが出せるところへ来たら，次々と翻訳処理を行つてゆく方法で，ソース・プログラム自体は必要最小限の部分だけしか記憶しない．これに対し，非逐次処理方式はソース・プログラムの一つのステートメントを一たん全部読み込んで，全体を見渡してからオブジエクト・プログラムを作成するという方法である．

1°　スタックを用いる方法　[1][5]他

この処理方式の根本原則はLIFO（Last−in−first−out）ということである．この処理方式では，ソース・プログラムを一時的に記憶しておくためにスタック（棚；stack，push−down，cellar）を利用する．通常各種演算子を保持するためのものと，変数，定数，および表示形関係（被演算数）を保持するためのものとの二種に別けられる．変数ないしは定数が読み込まれたならば，これに対応したstorage location を被演算数のスタックへ保持する．演算子が読み込まれたときには，演算子のスタックの一番上のものと比較し，翻訳処理を行うか，すなわちオブジエクト・プログラムを出すか，この演算子をスタックに保持するかは優先順位により決定される．この演算子の優先順位（hierarchy）を行列の形に整理したものが解読行列（translation matrix）である．

解　読　行　列

棚の一番上にある演算子 ＼ 新しく読み込んだ演算子	:=	(	+，−	×，／，÷，NEG	↑	)	;
:=	SR	SR	SR	SR	SR		PRP
(		SR	SR	SR	SR	PRD	
+，−		SR	PRD	SR	SR	PRP	PRP
×，／，÷，NEG		SR	PRP	PRD	SR	PRP	PRP
↑		SR	PRP	PRP	PRD	PRP	PRP

SR：stack and Read

PRP：Pop−up and Repeat

PRD：Pop−up and Read

すなわち，簡単に説明すると，新しい演算子が，演算子のスタックの一番上にある演算子に比べて，その優先順位が高い場合は，その演算子はスタックに保持（stack）され，低いあるいは同順位の場合には，スタックに保持されていた優先順位の高いあるいは同順位の演算子が飛び出し（pop-up）翻訳処理が行われるのである．

例］ $E := (A \times B + C \times D)$

演算子の棚	新しく読み込んだ演算子	オペランドの棚	オブジェクトプログラム
ϕ	E	ϕ	
ϕ	$:=$	E	
$:=$	(	E	
$:=$ (	A	E	
$:=$ (	×	E , A	
$:=$ (×	B	E , A	
$:=$ (×	+	E , A , B	$A \times B \rightarrow$ Acc.
$:=$ (	repeat	E , Acc.	
$:=$ (+	C	E , Acc.	
$:=$ (+	×	E , Acc., C	
$:=$ (+ ×	D	E , Acc., C	
$:=$ (+ ×	)	E , Acc., C , D	Acc. → Temp 1 $C \times D \rightarrow$ Acc.
$:=$ (+	repeat	E , Temp 1 , Acc.	Temp1 + Acc. → Acc.
$:=$ (−	repeat	E , Acc.	
$:=$	;		Acc. → E
ϕ		ϕ	

なお，穴倉方式と呼ばれる，優先順位の高いあるいは同順位の演算子が読み込まれたときには，これをスタックし，低いものが来たときに初めてスタックの上の方から翻訳処理するという方法，前述の如く，文法構造の状態をスタックする方法等スタックを用いる方法も数多くある．

2° カッコを挿入する方法 ［2］

この方法は，FORTRAN の算術式を処理するために考えられた非逐次翻訳方法である．オブジェクト・プログラムが，作られるまでには，何回ものscan，変換，整理が行なわれる．この方法の特徴は， operator の強さを表わすためにカッコを挿入して， 算術式を operator と operand の 組からなるstringに分解して，それぞれの組について，位置づけ（レベリング）を行い，そのレベルをしらべることによつて，計算順序を決めて行くことと，算術式の中に共通な部分算術式が現われたときには，その部分の計算を1度ですますような最適化が，行なわれることである．

翻訳の過程は，次の5段階に分けられる．

ⅰ） Reduction

ⅱ） 正 規 化

ⅲ） レベル分析

ⅳ） 最 適 化

Ⅴ） コンパイレイション

以下これらについて，その概要を述べる．

1） Reduction

算術式の定数，添字つき変数を，それと mode が同じで，その算術式に現われない name で置き換える．

2） 正 規 化

Operator の優先順位によつて，Operator と Operand との間に，カツコおよび，記号を挿入する．その方法は，次のとおりである．

ⅰ） $A**B \rightarrow A)**(\oplus B$

$A*B \rightarrow A))*(**(\oplus B$

$A/B \rightarrow A))/(**(\oplus B$

$A\pm B \rightarrow A)))\pm(*(**(\oplus B$

ⅱ） 算術式の両端と算術式に現われたカツコについては，次のようにする．

左端，(→ +(*(**(⊕

右端，) →)))

（例） $-A+B*C$

という算術式を考えると，次のような標準形に変換される．

$-(*(**(\oplus A)))+(*(**(\oplus B))*(**(\oplus C)))$

3） レベル分析

標準形の operator ($\diamondsuit_i$) と operand (φ_i) との一組を考えて，それの算術式における位置を示すためにレベル番号(C_i)を付け，triple (C_i, $\diamondsuit_i$, φ_i) からなる string $\pi(\phi)$ (production) を作る．$\diamondsuit_i$, φ_i を次のように定義する．

$\diamondsuit_i$ が null で，φ_i =) あるいは，

$\diamondsuit_i \in \{+, -, *, **, /, \oplus\}$ で，φ_i が name または (

Productionのつくり方

$\phi_i =($	$\phi_i =$ name	$\phi_i =)$
$\pi_i \to \pi_i (C_i, \diamondsuit_i, N_i)$	$\pi_i \to \pi_i (C_i, \diamondsuit_i,$ name$)$	$\pi_i \to \pi_i$
$N_{i+1}=N_i+1$	$N_{i+1}=N_i$	$N_{i+1}=N_i$
$C_i=N_i$	$C_{i+1}=C_i$	$C_{i+1}=K_i'$
$A_{i+1}=A_i+1$	$A_{i+1}=A_i$	$A_{i+1}=A_i-1$
$K_{i+1}=(K_i, C_i)$	$K_{i+1}=K_i$	$K_{i+1}=\overline{K}_i$

(K_i, C_i)は，数列KにC_{i+1}を続けることを意味し，K_i'は，Kの最後の要素，$\overline{K}_i$は，Kから最後の要素を取り去つたものを意味する．

ただし，$N_1=1$，$C_1=A_1=0$，$K_1=\pi_1=$ null で，$\pi_i \to \pi_i E$と変換されたとき，それをπ_{i+1}とおく．あるレベル番号Cを持つ tripleの集合をレベルCの segment (S_C)，それに属すtrlpleの数を lengthと言う．

（例）前の例を用いると，次のようなproductionが得られる．

$\diamondsuit_i$ ϕ_i	−(	*(	**(	⊕A	)	)	)	+(	*(	**(	
i	1	2	3	4	5	6	7	8	9	10	
N_i	1	2	3	4	4	4	4	4	5	6	
		⊕B	)	)	*(	**(	⊕C	)	)	)	
		11	12	13	14	15	16	17	18	19	20
		7	7	7	7	8	9	9	9	9	9

$\pi(\phi)=$(0, −, 1)(1, *, 2)(2, **, 3)(3, ⊕, A)(0, ±, 4)

(4, *, 5)(5, **, 6)(6, ⊕, B)(4, *, 7)(7, **, 8)(8, ⊕, C)

4） 最 適 化

i）標準形になおしたときには，多くのカツコを挿入したが，最適化の第1段階は，これらの余分なカツコを削除する．それは，$\pi(\phi)$を後からながめて行つて，triple iが，length 1 の segmentで，$\diamondsuit_i$キーならば，その tripleを削除し，ϕ_iをすぐ前の tripleの3番目の要素に置き換える．

（例） $\pi(\phi)=$(0, −, A)(0, +, 4)(4, *, B)(4, *, C)

ii）$\pi(\phi)$を低レベルの順序にならべかえる．これを$\overline{\pi}(\phi)$とすると，

$\overline{\pi}(\phi)=S_1 S_2 \cdots S_L$

$S_C=(C, \diamondsuit_C^1, \phi_C^1) \cdots (C, \diamondsuit_C^{\lambda_C}, \phi_C^{\lambda_C})$ である．

iii) 部分算術式の処理

高レベルのsegment S_Lからはじめて，すべてのS_j ($j<i$, $i \leqq L$)について，$S_i=S_j$かどうか調べる．$S_i=S_j$のときには，S_jを$\overline{\pi}(\phi)$から削除し，他のsegmentで$\varphi_k=j$となつている所を$\varphi_k=i$にする．

(例)　$\Phi=A*(B*C)+\mathrm{SINF}(A*(B*C))$

$\overline{\pi}(\phi)=$(0,+,1)(0,+,14)(1,*,A)(1,*,7)(7,*,8)(7,*,C)
(14,⊕,SINF)(14,⊕,16)(16,*,A)(16,*,22)(22,*,B)(22,*,C)

$\overline{\overline{\pi}}(\phi)=$(0,+,16)(0,+,14)(14,⊕,SINF)(14,⊕,16)(16,*,A)
(16,*,22)(22,*,B)(22,*,C)

iv) それぞれの計算機の特徴を考慮して，できるだけレジスタ間のデータのやりとりを少くするように，すぐ後のsegmentとの関係をみて，tripleを適当に入れ換える．

5)　コンパイレイション

最適化されたproductionを，オブジエクトプログラムへ翻訳する．この翻訳は，高レベルのsegmentから行なわれ，segment内では前の方から行なわれる．

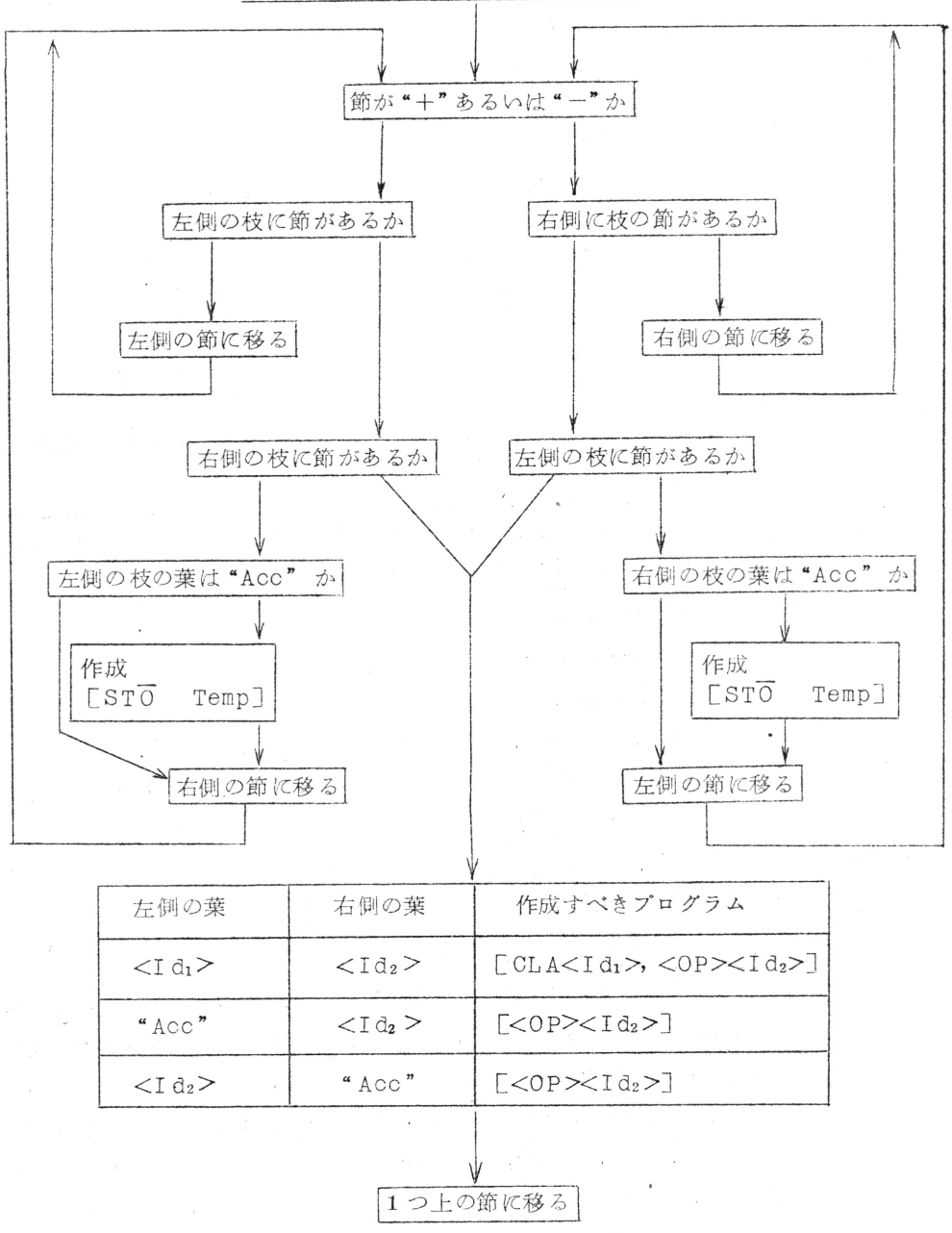
3° 木構造を用いる方法　[6]

この処理方式は通常のスタック方式を用た場合よりも多少なりとも効率のよいオブジエクト・プログラムを作成することを目的としている．

先ずソース・プログラムを木構造に変換する．この場合，その節(node)に演算子を，葉(leaf)に被演算数をもつ木構造とする．

木構造が形成されたならば，次の流れ図(flow chart)に示すアルゴリズムに従つて，その木構造の根(root)から始めることにより翻訳処理を行うのである．

木構造に基づく翻訳処理のアルゴリズム

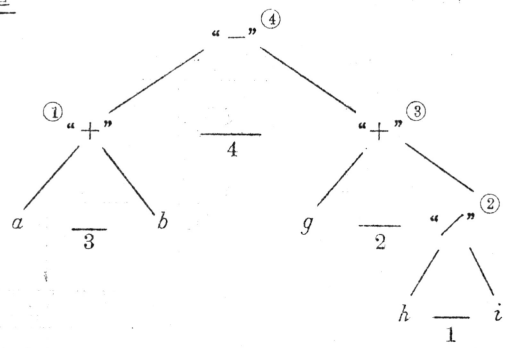


左側の葉	右側の葉	作成すべきプログラム
<Id$_1$>	<Id$_2$>	[CLA<Id$_1$>, <OP><Id$_2$>]
“Acc”	<Id$_2$>	[<OP><Id$_2$>]
<Id$_2$>	“Acc”	[<OP><Id$_2$>]

1つ上の節に移る

例]　$a+b-(g+h/i)$

木　構　造



ただし①～④は部分木（sub−tuee）の形成順序を，$\overline{1}$～$\overline{4}$は翻訳処理の行われる順序を示す．

通常のスタック方式を用いた場合に作成されるプログラム	木構造を用いた場合に作成されるプログラム
$a+b \rightarrow$ Acc.	$h/i \rightarrow$ Acc.
Acc. → Temp 1	Acc. $+g \rightarrow$ Acc.
$h/i \rightarrow$ Acc.	Acc. → Temp 1
Acc. $+g \rightarrow$ Acc.	$a+b \rightarrow$ Acc.
Acc. → Temp 2	Acc. − Temp 1 → Acc.
Temp 1 → Acc.	
Acc. − Temp 2 → Acc.	

4°　Polish表示に翻訳する方法　[7]

このPolish表示（Polish notation）はポーランドの論理学者Lukasiewicz によつて導入された表示法であるが，この表示法が算術式においてカツコを必要としない（parenthesis−free）表示法であることから，ソース・プログラムを一たんPolish表示に変換しておけば，オブジエクト・プログラムに翻訳することが容易になるという特徴を備えている．特に，KDF9，B−5000のようにpush−down memory を金物でもつている機械では，この表示法を直接受け入れてくれるのでコンパイルは非常に簡単になる．

このPolish表示に翻訳する処理方法は，演算子用のスタックのみを用いて逐次処理するのであるが解読行列は1°の場合と全く同じでよい．

翻訳処理手順

1）被演算数が読み込まれた場合は，直ちにこれをアウト・プットする．

2）演算子が読み込まれると，スタックの最上位の演算子とその優先順位を比較し，高い場合には，スタックする．同順位あるいは低い場合には，スタックの最上位が読み込まれた演算子の優先順位に比し低くなるまで，スタック内の演算子を順次アウト・プットする．その後，その演算子をスタックする．

3）開きカッコが読み込まれた場合には，これを直ちにスタックする．

4）一方，閉じカッコが読み込まれた場合には，それに対応する開きカッコが表われるまで順次スタック内の演算子をアウト・プットし，その開きカッコをスタックから取り除き，次を読み込む．

5）ステートメントの終止記号“；”が読み込まれた場合には，スタック内の演算子を順次アウト・プットすればよい．

例　$g := a+b\times(c-d)\div e-f$;

	棚	読み込んだもの	オブジエクトプログラム		棚	読み込んだもの	オブジエクトプログラム
1	ϕ	g		12	:=, +, ×, (	repeat	“−”
2	ϕ	:=	g	13	:=, +, ×	÷	
3	:=	a		14	:=, +	repeat	“×”
4	:=	+	a	15	:=, +, ÷	e	
5	:=, +	b		16	:=, +, ÷	−	e
6	:=, +	×	b	17	:=, +	repeat	“÷”
7	:=, +, ×	(		18	:=	repeat	“+”
8	:=, +, ×, (	c		19	:=, −	f	
9	:=, +, ×, (	−	c	20	:=, −	;	f
10	:=, +, ×, (, −	d		21	:=	repeat	“−”
11	:=, +, ×, (, −	)	d	22	ϕ		“:=”

オブジエクトプログラム

$g\ a\ b\ c\ d-\times e\div+f-:=$

3.2.2　ブロック構造他　[3]

1)　Block構造

2)　ProcedureのRecursive Call

3)　Array ElementsのDynamic Allocation

4)　Own Variable

5)　Call by ValueとCall by Name

6)　Type

以上のような点が問題になると思われるので，これらの意味を考え，具体的な処理の一例として，Whetstone Compiler [3]をこの点に限定して紹介する．

1)　Block構造

ALGOLが，block を，そのプログラムの最小の構成単位とすることは特徴の一つである．ある一つのblockのheadでdeclareされたquantityは，そのblockの内部でのみ意味をもつ．

あるblockが，他のblockをその内側に含んでいるとき，後者は，前者より深い部分にあると考えて，それぞれのblock に深さの度合を表わす正の整数を対応させるものとすると，深さnのblockが実行されるときには，意味をもつquantityは，深さ$n-1$，$n-2$，$\cdots$ 2，1のblockにおいてdeclareされたものだけであり，しかも，それらのblockは，より大きい深さをもつものを，それぞれ内部に含む関係にある．

Whetstone Compilerでは，ALGOLで書かれたプログラムは，one-passで中間言語(object program)に変換され，それが，Control Routineによつて解読されながら実行される．

object programに変換されたとき，variable nameは，自分がdeclare されたblockのblock number （上に述べた深さ）と，その中での相対番号をもち， labelもobject program上でその labelが相当する番地の他に，自分がdeclareされたblockのblock numberを持つている．

blockとprocedureは，対等に扱われる．

新しくblockに入るときは，その新しいblockのblock numberと，その中でdeclareされたlocal variableの個数をparameterとして持つobject program命令（BE(n, a)，procedureのときはPE(n, a)mで，mはパラメターの数）が置かれる．

object programがControl Routineによつて実行されるときは，すべての仕事はStackを用いて行なう．新らしいblockに入るごとに，そのblockの前後関係を示すlink data（3語からなる）をまずstackし，その上に，そのblockでdeclareされた

local variableのための場所をstackし，さらにその上の部分のstackを用いながら現在のblockでの作業を実行する．

このblockがendに到達して完了すれば，そのblockのためのstackはlink も共に取り去つてしまう．endに達する前に他のblockに入つてしまえば，また同じようにstackを重ねて行く．このようにすれば，現在validなquantityは，このstackの下積みになつているものの中にあり，それらのうちで，link dataをたどつて“含む”関係にあるblockのquantityのみが現在validであるといえる．

例．

```
begin real a1;
   procedure Q2;
   begin real b2, c2;
         ...
   L2 : begin real d3;
         procedure R4;
         begin real f4, g4;
            ...
         L4 : g4 := 0
         end;
      L3 : begin real h4;
            ...
         M4 : R4;
            ...
         end
      end
   end;
P1 : begin real i2, j2;
         ...
   P2 : begin real l3;
      N : Q2;
         ...
      end
   end
end
```

上のALGOLプログラムのobject programは，次のようになる．

```
loc.   op.    par.              rem.
  0    CBL              call program block
  1    UJ      603
  4    BE     (1,1)     program block
  7    UJ      403
 10    PE     (2,2)        procedure Q2
 14     :
  :     :
100    CBL                    call block L2
101    UJ      402
104    BE     (3,1)           block L2
107    UJ      205
110    PE     (4,2)              procedure K4
114     :
  :     :
200    TICO                     g4:=0
201    ST     (4,4)
204    RETURN                   K4 end
205    CBL                    call block L3
205    UJ      401
209    BE     (4,1)           block L3
212     :
  :     :
300    CFZ     110             call function K4
303     :
  :     :
400    RETURN                   block L3 end
401    RETURN                block L2 end
402    RETURN             procedure Q2 end
403    CBL                call block P1
404    UJ      602
407    BE     (1,2)        block P1
410     :
  :     :
500    CBL                    call block P2
501    UJ      601
504    BE     (2,1)           block P2
507    CFZ      10            call function Q2
510     :
  :     :
600    RETURN                   block P2 end
601    RETURN             block P1 end
602    RETURN          block end
603    FINISH
```

上の object program の中で …の所は，新たに block に入る作業を含まないようなプログラムであると仮定している．また 100, 200, … の loc.は便宜上書いたものである．

この object program が実行されたときの loc. 200 のところにおける stack の状態は図1のようになる.

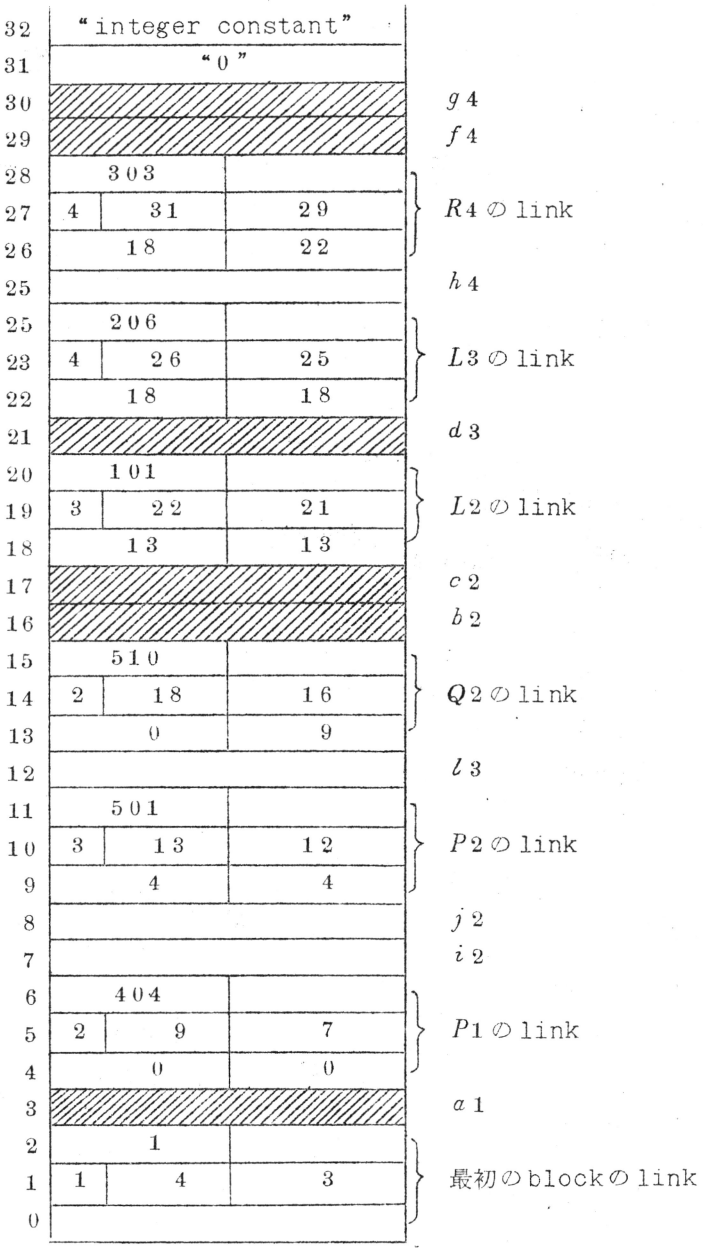


図1　　loc. 200 における stack

link data 3語の構成は

RA		m. c. l.
BN	WP	FP
S C		D C

となつており，D C（dynamic chain）をたよりにしてS C（static chain）をたどれば，現在 valid なものを知ることができる．この例で，現在 valid（accessible）であるものは斜線部分である．

2） Procedure の Recursive Call

Procedure の中で，さらに自分自身を call することがあると，working space などが，用済みにならないうちに，また同じ作業をくり返さなければならない．またこの call は，時によつて何回くり返されるかわからないので，静的な，あるいは，平面的な allocation を行つたのでは，処理がむつかしい．実例としてあげている Whetstone Compiler のような，そのつど stack をつくりあげる方式では，簡単に扱える．

例 2. 正整数の階乗

```
real procedure Fac(n); integer n;
  begin
    if n≧1 then go to L1;
    Fac:=1;
    go to L2;
  L1: Fac:=n×Fac(n−1);
L2: end
```

この procedure が，たとえば Fac(3) と call されたとすると，内部で Fac(2)，そのまた内部で Fac(1) が call されるので，一番内側の Fac(1) の時の stack の状態は，三重の層をなすが，Fac(1)，Fac(2) と計算が完了するごとに，一層ずつ取り除かれる．

3） Array Elements の Dynamic Allocation

array declaration の中の bound pair が，arithmetic expression であるために，その値は実行時にならなければきまらない．従つて，array element に対する割り当ては，array declaration が現われる所で，そのつど bound pair のそれぞれの値を計算して行うことになる．Whetstone Compiler では，quantity が declare された所で，その分だけの space を stack の上に積み上げている方式を取つているので array element の数が，大きくふくれたり，小さくなつたりしても問題にならない．

例　次のようなdeclaration partをもつblockがある．このblockのblock numberは2であるとして，i，jはこのblockを含むblock number 1のblockで，declareされたinteger variableで，それぞれaddress(1, 4)(1, 5) を与えられているとする．

```
begin integer M, N;
      real a;
      array A, B[i:j−3, 0:1];
     ...
     ...
end
```

上のarray declarationのところのobject programは次のようになる．

loc.	op.	par.	rem.
0	BE	(2, 5)	Block Entry
3	TIR	(1, 4)	i
6	TIR	(1, 5)	j
9	TIC	"3"	
16	—		$j-3$
17	TIC0		
18	TIC1		
19	MSF	(2, 7), 2	(2, 7)=B　2個

このobject programが実行されるとき，i，jの値は，その外側のblock できまつているはずで，今$i=-1$，$j=4$となつているとして，19のMSFの直前における stackの状態は図2のようである．

MSF(make storage function)は現在のstackの上部にあるbound pairから，array elementをreferするための関数をつくり，array identifier のためにある番地に，その関数のある場所とarray elementの最初のものがある場所と，0番目のelementのある場所を書き込む作業をする．

MSFの完了後のstackの状態は図3のようになる．

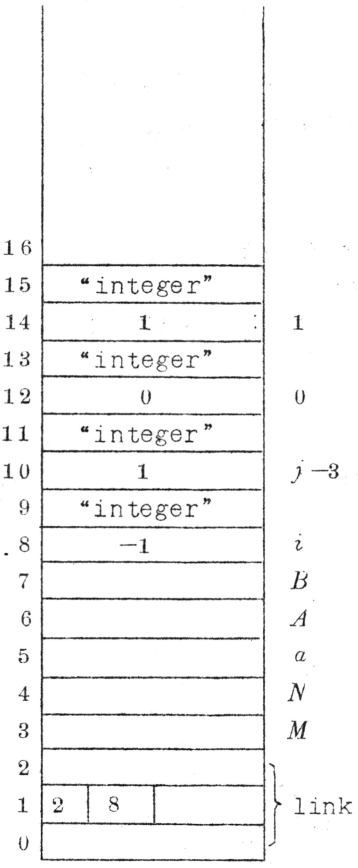


図 2

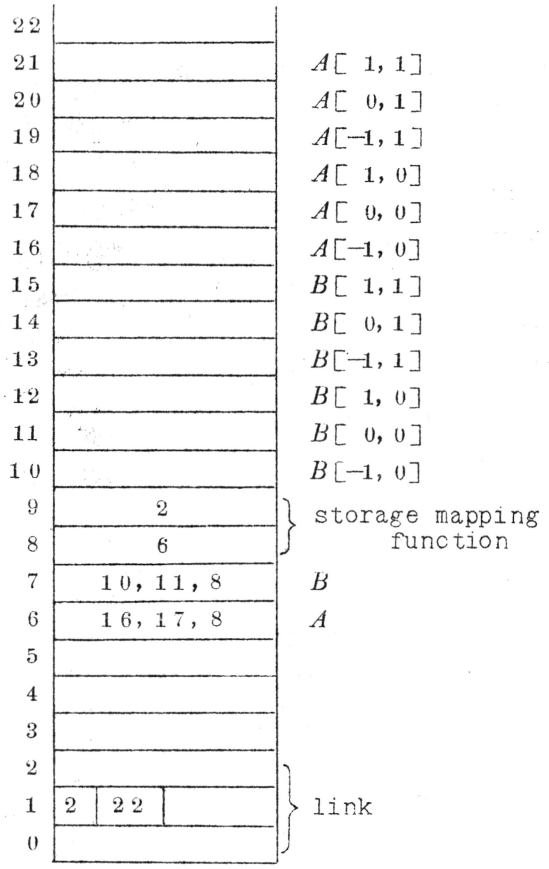


図 3

このようにして，allocate されたaddressをたとえば

$B[0,1] := 0$

というようにreferするときのobject programは次のようにする．INDAはBのための番地(2,7)を見て，storage mapping functionと，$B[0,0]$の番地が11であることとを知り，$B[0,1]$が13(11+1×2+0=13)であると決定することを要求する命令である．

loc.	op.	par.	rem.
0	TRA	(2,7)	(2,7)=B
3	TIC0		
4	TIC1		$B[0,1]$
5	INDA		
6	TIC0		
7	ST		$B[0,1] := 0$

4) Own variable

declarationにおいて，ownと頭書きのついたquantityは，次にそのblock に入るときまで，現在の出口における状態を保存しなければならない．他の quantity は，そのdeclarationがあるblockの出口を出してしまえば，意味を失うことから考え合せて，own quantityは，プログラムの一番外側のblockでdeclaration がなされたかのように扱えばよい．

Whetstone Compilerでは，blockの出口へ来ると，そのblockのためのstackはlinkと共に取り去つてしまう方式であるので，own variableは，stackの中へ積み込んでしまうわけにはいかない．

object programをつくる段階で，そのプログラム全体を通じて用いられたown variableの総数を数えあげ，実行段階に入るときに，まずstack の底を“あげ底”にして，その上のstackを用いて作業を開始し，ownが出てくるごとに，“あげ底”の下の部分を割り当てている．

own variableの総数は，compileの段階で決定される必要があるので，own arrayに関しては，そのbound pairがinteger constantであることを制限条件としている．

またownでないvariableは，declareされると，そのつどstackの上に積みあげたのに対して，own variableは，取り除かれることがないのであるから，allocationも最初の一回だけでよい．次の例にあるAOA(Avoid own array)は，そのcontrolのために置かれた命令である．own variableに対しては，block numberが，いくつの時でも，block number 0の扱いをする．

例　own real array $A[1:4]$;

loc.	op.	par.	
0	AOA	(15)	二度目以後は，15へjump
3	TIC1		
4	TIC	“4”	
11	MOSF	(0,5),1	$(0,5)=A$
15	...		

array identifier Aのための番地が(0,5)であるとした．

5) Call by valueとCall by Name

call by valueはprocedureのbodyに入る前にvalue assignmentをする部分をつけるだけで単純に考えてよい．call by nameはactual parameter の形によつて，procedure bodyのformal parameter に相当する部分の作業内容そのものが変るので，問題になつて来る．

procedure bodyのcompileは一回だけ行なつて，実質上，そのprocedureがcallされるごとにactual parameterの形に従つてrecompile したかのような効果を出す工夫が必要になる．

しかも，actual parameterの中に含まれるparameterは，そのcall statementが書かれた位置において valid なものであるが，callされるprocedureのbodyの位置では validとは限らない．

Whetstoneでは，subscripted variable，expressionなどのactual parameterは，blockの形式をもつたsubroutineにして，CF(call function)命令の直前におき，これをformal parameterの場所でcallする処理方法をとつている．

例　procedure Pのcall statement

$P(-A, 5, B+C[I])$

のobject programは次のようにする．

ここでこのcall statementのおかれているblockのblock number は2であるものとし，A, B, C, IなどのaddressはA, B, C, Iで表現し，procedure Pのある番地もPで表現しておく．

```
loc.    op.     par.
 0      UJ      (44)        Call Functionへjump
 3      BE      (3,0)       Block Entry
 6      TRR     A           −Aのための subroutine
 9      NEG                     block number 3
10      EIS                 End Implicit Subroutine
11      "5"                 constant 5
17      BE      (3,0)
20      TRR     B           B+C[I]のための subroutine
23      TRA     C
26      TIR     I
29      INDR
30      +
31      EIS
32      PSR     (17),−      Parameter Subroutine
36      PIC     (11),−      Parameter Integer Constant
40      PSR     ( 3),−
44      CF      P, 3
48      ...
```

これに対し，procedure P のdeclaration

procedure $P(u, v, w)$;
...
...

のbodyがobject programになつたとき，u, v, w に対するreferは

TFR* (Take Formal Real)
TFI* (Take Formal Integer)

などの命令によつて表現される．たとえばTFR u，という命令がAのためのsubroutine callを要求することになる．

actual parameterのsubroutineをblock形式にしたために，actual parameterが自分自身をcallしても，大丈夫である．

例 real procedure $Q(n)$; integer n;
begin ...
$P(A+B\times Q(n-1))$;
...
end

(* Procedure declarationのformal parameterのtypeをすべて，specification partでdeclare することを制約条件としている．)

あるformal parameterをactual parameterとして用いるときのために，PF (Parameter Formal)というobject program命令をおく．

6) type

ALGOLの文法を忠実に守ろうとすると，typeに関する取り扱いが，意外にわずらわしい問題となつてくる場合がある．

procedure declarationのformal parameterのtypeの問題もその一つといえる．actual parameterにdependしてきまつてくるので，compileの段階では，全く未定義である．(Whetstone Compilerでは，procedure declarationのspecification partにformal parameterに関するfull specifyを要求している．)一般には，formal parameterは，実行時にactual parameterによつて，どのようにcallされるかによつて，どのtypeも受け入れるようにしておかねばならない．

演算中のtypeの取り扱いも，やつかいな問題である．+, −, ×, /では，二つのoperandのtypeに従つて，結果が規定されているだけであるのに，÷は，結果がintegerであることを規定する以外に，二つのoperandが共にinteger であることを規定しているなど，その例の一つである．

4. その他

使い良いコンパイラを作るためには，ただソースプログラムが翻訳できるという他に，デバックのこと，誤りの処置，他のシステムとの関連などについても十分考えに入れておかなければならない．そのために，コンパイラが厖大になつたり，オブジエクトプログラムの効率が多少落ちてもある程度は止むを得ない．

4.1 誤りの検出

ソース・プログラムの誤りを検出するのはコンパイラの一つの重要な役割りで，このためにコンパイラはかなり多く部分を割かなければならない．

プログラムの誤りには文法上の誤りとプログラム論理の間違いとあり，コンパイラは文法上の誤りについては手をかければかなり検出できるが，後者の誤りの検出にはまつたく無力でほとんどできないといつてよい．

4.1.1 ソースプログラムの誤り

ソースプログラムのエラーを検出すると，その位置と原因を印刷するが，その原因については往々にして見当ちがいのメッセージが出ることがあるので，エラーのあつたステートメントやブロックの一部または全部を印刷もすると良い．エラーを起した位置については：4.1.2 b) と同じでよく，実行時の場合よりはずつと簡単である．

誤りを検出したときの処置にもいくつかあつて

a) 誤りを発見しだいコンパイルをやめる．

これはコンパイラにとつては最も楽な方法であるが，一回のコンパイルでエラーが一つしか見つからないのでコンパイルの回数が増えて得策ではない．

b) そのパスだけは終らせる．

誤りのあつたステートメントまたはブロックはさらに検査を続けても，始めの誤りが原因で次々とエラーとなつて無意味なことが多いので，そのステートメントやブロックは無視して，ともかくそのパスだけは終らせ，それ以後もできるだけ誤りは検出しておく．

c) 何らかの方法でプログラムを矯正して，オブジエクトプログラムを作り出す．はなはだしいエラーのときは無意味だが，オブジエクトプログラムが簡単に訂正できるとき便利なことが多い．

4.1.2 実行時の誤り

負の数の平方根や対数の計算とかオーバーフロー，行き先のないGO TO(COBOL)などはコンパイル時には検出できないので，どうしても実行時にチエツクしなければならない．

エラーがおきたらエラーメッセージを出してその仕事をやめるとか，何らかの処置（たとえば $\sqrt{x}=\mathrm{sign}(x)\cdot\sqrt{|x|}$，IBM7090 FORTRAN II）をして先へ進むとかするわけであるが，そのエラーメッセージにはやはりエラーの起きた場所と原因を何らかの方法で知ら

せなければならない．その場所については

a）オブジエクト・プログラムでエラーのわかつた番地とか，サブルーチンの中ではそのサブルーチンからの戻り番地などを印刷する．これでソース・プログラムとオブジエクト・プログラムの対応表があれば一応わかるわけだが，ローデイングのとき任意にrelocateしてあればわからなくなつてしまう．

b）ソース・プログラムのどこでエラーがおきたかわかるようになつていれば一番良いが，そのための情報を実行時に持つていなければならないのでかなり面倒で，効率も悪くなる．

FORTRAN：上の一番近いステートメント番号から何行目のステートメントかを示す．

ALGOL：ALGOLの場合一番困難であるが，しいていえば一番近いlabelのついたブロックまたはprocedureから何番目のブロックの何番目のステートメントということであろうか．しかしこのlabelも必ずしもユニークではないので厳密に言えばユニークになるまでさかのぼつてlabelを列挙しなければならない．

COBOL：普通はsequence numberでことたりる．ただし，INCLUDEを使うとユニークでなくなることがあるので，さらにPARAGRAPH名とSECTION名を出せば完全である．

4.2 デバッグ

コンパイラはオブジエクト・プログラムのデバッグについても十分な配慮が必要で，そのデバッグのやり方は，モニターとかローダーの性能によつて異なつてくる．

a）ソースプログラムを書くときには特にデバッグのことを考えなくとも，デバッグしたいプログラムをロードする際に，中間結果を印刷したい変数の名前，位置（ソース・プログラムの場所），条件（たとえば，1回おきとか，最初の3回だけとか，条件を満足するときだけ印刷する），印刷のしかたなどを指定する．

このためには，オブジエクト・プログラムをローデイングするときまで変数の名前やタイプなどの入つた辞書を持つていなければならない．

この方法では主役はモニターとローダーなので，コンパイラはモニターにデバッグ用の情報を与えてやればよい．

b）プログラマがソース・プログラムを書くときにデバッグのことを考えて適当なところにデバッグ用のステートメントを挿入しておく．デバッグ終了後実際の仕事をするときにはこのステートメントは無視されるかローデイングの際に除かれる．

この方法では予期しなかつた場所が印刷したくなつたときは不便だし，もしコンパイルをやり直さなければとり除けないときは意味がない．

c）もしコンパイラがデバッグのことを全然考えに入れないと，ソース・プログラムとオブジエクト・プログラムの対応表を調べなければならないので，機械語を知らない者に

はデバッグできずコンパイラの趣旨に反する．

とにかくソース・プログラムの変数の名前を使つてデバッグできるようになつていることが望ましい（Source language debug）．

4.3　オブジエクトプログラムの最適化

コンパイルされたオブジエクトプログラムを熟練したプログラマが十分時間をかけて作つたプログラムより効率を良くすることは不可能であろうが，コンパイル時にある程度手間をかければかなり良いプログラムにすることは可能である．

普通コンパイルするときには1つのステートメントだけしか見ないので，最適化といつてもせいぜい，1つのステートメント内で同じ計算をくりかえさないようにしたり，冗長な命令を除くといつた程度のことしかできにくい．前後いくつかのステートメントを見ればもつとよくなるはずであるが非常に難しく，現状では最適化についてはほとんどプログラマの腕に委ねられている．

この方法については条件付翻訳のための巧妙なマクロ命令を作るとか，stack に色々な目印をつけるなど多く考えられるがここでは省略して，いくつかの例をあげるにとどめる．

例1

$$X=(A+B)*Y+C*Z**(A+B)$$

$(A+B)$を2回計算しないで実質的に

$$T=(A+B)$$

$$X=T*Y+C*Z**T$$

例2　常数の計算はコンパイル時に

$$y:=\mathrm{sqrt}(2.0/3.14159927)/x$$

は

$$y:=0.7978845/x$$

例3　因数分解

$$Y=C1+C2*X+C3*X**2+C4*X**3$$

は

$$Y=C1+(C2+(C3+C4*X)*X)*X$$

例4　変換の回数を減らす

COBOLの算術式でBがDISPLAYだとすると

COMPUTE $A=B+(C+B)*0.5$

は，Bを2回ともDISPLAYからCOMPUTATIONAL の変換をせずに，コンパイラがCOMPUTATIONALのworking－storage Wを作つて

MOVE B TO W

COMPUTE $A=W+(B+W)*0.5$.

とする.

例5　冗長な命令を減らす

IF(a) m_1, m_2, m_3

は一般に

```
        JZE   m2          (Jump on Zero)
        JPL   m1          (Jump on Plus)
        JUC   m3          (Jump unconditional)
```

であるが場合によつて

(イ) IF(a) m_1, m_2, m_1

m_2 ····

```
        JNE   m1          (Jump on Non-zero)
```

(ロ) IF(a) m_1, m_2, m_3

m_1 ····

```
        JZE   m2
        JMI   m3          (Jump on Minus)
```

4.4　他のシステムや言語との関係

どんな計算機システムでもプログラム言語がコンパイラ1個だけということはまずないので，1つの言語で使用するデータが他の言語やプログラムシステム（たとえばSORTとかLPなど）のデータを共通に使えるようになつていることが必要だし，連続して異なつた言語で書いたプログラムやプログラムシステムの作業を行う際にはデータの受授がスムーズにできなければならない．これらのことは通常モニターがやつてくれるので，コンパイラがモニターの監督下に入つて他と矛盾しないようになつていればそれでよい．

すでにできているルーチンがあるとか，またどうしても一部分だけ他の言語でプログラムを作つた方が効率がよいといつたことがしばしばあつて，異つた言語が他のプログラムの中に入ることがある（例，COBOLのENTER）．

このとき一番問題になるのがidentifier の連絡をどうするかということで，たとえばCOBOLの

```
        ENTER  FORTRAN.
```

があればFORTRANと連絡をとるCOBOLのidentifierは両方の言語に矛盾しないようにそのデータはすべてword単位になつていて，identifier もユニークで修飾の必要がなく，しかも6文字以内でなければならないといつた制限をつけて逃げるとか，コンパイラが特別に名前をつけかえてあとは下のアセンブラにまかせるといつたことをしなければなら

ない．

ようするにコンパイラ（他のプログラムでも同様であるが）単独でしか動かないというのは協調性に欠けていてよろしくない．

4.5　コンパイラの保守

コンパイラ言語の文法そのものの改訂，コンパイラ自身の間違いの発見，もつと効率のよいプログラムの開発，あるいは使用者の特殊性や好みなどによつて，コンパイラの訂正が必要になることは多い．したがつて，コンパイラ自身を訂正する標準的な方法が確立されていることは必要である．

そのためには，コンパイラの動きのいろいろなレベルでのフロチヤートや詳しい記述，各種サブルーチンやテーブルの機能や使い方，コンパイラを書いた言語のステートメント毎の註釈などを豊富につけて，読みやすくなつていなければならないし，またコンパイラのプログラムもできるだけ手品師的な巧妙なプログラムは使わず，標準的なテクニックのみを用いてできているとよい．

これらはコンパイラを作るものにとつてははなはだやつかいなことであるが，人の出入のはげしい今日では作つた人にしかわからないというコンパイラは融通性に欠けて作成者側にとつても好ましくない．

実際にプログラミングした人以外でもそれ程苦労せずに保守できるようになつていることもコンパイラの必須条件である．

5．あとがき

近年コンパイラを作るのが比較的簡単な syntax directed compiler ということがしばしば言われるようになり，その方法によるコンパイルの例もいくつか報告されているが［9，10，11］，現状ではまだオブジエクトプログラムの効率の点で劣るので広くは採用されていない．しかし，文法が多少変つても syntax そのものを入力と考えれば簡単に修正できるといつた長所があり将来有望であるがここでは経験も資料もまつたく不足なので省略した．

コンパイラをコンパイラの言語で書くことも［8］，興味があるがいまのところ実用の域に達しているものはない．しかしいいものを早く完成させてコンパイラを作る作業を楽にしたいものである．

現在最も多く使われている FORTRAN とか ALGOL とか COBOL といつたプログラム言語はみな読みやすさとか，コンパイラの作りやすさといつたことに重点をおいて作られているが，今後はプログラマが間違いを犯しにくく，しかも見てわかりやすいプログラム言語（たとえば表にするとか，絵を書くといつたような）を研究する必要があるだろう．

6. 参考文献

[1] Samelson, K., Bauer, F. L. "Sequential formula translation" Comm. ACM 3 (Feb. 1960) 76～83.

[2] Sheridan, P. B. "The arithmetic translator－compiler of the IBM FORTRAN Automatic Coding System" Comm. ACM 2 (Feb. 1959) 9～21.

[3] Randell, B., Russel, L. J. "ALGOL 60 Implementation" Academic Press, 1964.

[4] Arden, B., Graham, R. "On GAT and the construction of translator" Comm. ACM 2 (July 1959) 24～26.

[5] 森口繁一 "ALGOL入門" 日科技連 1962.

[6] Anderson, J. P. "A Note on Some Compiling Algorithms" Comm. ACM 7 (March, 1964) 149～150.

[7] Randell, B., Russel, L. J. "Single－Scan Techniques for the Translation of Arithmetic Expression in ALGOL 60" J. ACM 11 (April 1964) 159～167.

[8] Garwick, J. V. "GARGOYLE, A Language for Compiler Writing". Comm. ACM 7 (Jan. 1964) 16～20.

[9] Conway, M. E. "Design of a Separable Transition－Diagram Compiler" Comm. ACM 6 (July 1963) 396～408.

[10] Irons, E. T. "A Syntax Directed Compiler for ALGOL 60" Comm. ACM 4 (Jan. 1961) 51～55.

[11] Floyd, R. W. "A Descriptive Language for Symbol Manipulation" JACM 8 (1961) 579～589.

[12] 和田英一 "COBOL Compilerの実験報告書" 日科技連 1964年3月

本 PDF ファイルは 1965 年発行の「第 6 回プログラミング―シンポジウム報告集」をスキャンし、項目ごとに整理して、情報処理学会電子図書館「情報学広場」に掲載するものです。

この出版物は情報処理学会への著作権譲渡がなされていませんが、情報処理学会公式 Web サイトの https://www.ipsj.or.jp/topics/Past_reports.html に下記「過去のプログラミング・シンポジウム報告集の利用許諾について」を掲載して、権利者の捜索をおこないました。そのうえで同意をいただいたもの、お申し出のなかったものを掲載しています。

過去のプログラミング・シンポジウム報告集の利用許諾について

情報処理学会発行の出版物著作権は平成 12 年から情報処理学会著作権規程に従い、学会に帰属することになっています。

プログラミング・シンポジウムの報告集は、情報処理学会と設立の事情が異なるため、この改訂がシンポジウム内部で徹底しておらず、情報処理学会の他の出版物が情報学広場 (=情報処理学会電子図書館) で公開されているにも拘らず、古い報告集には公開されていないものが少からずありました。

プログラミング・シンポジウムは昭和 59 年に情報処理学会の一部門になりましたが、それ以前の報告集も含め、この度学会の他の出版物と同様の扱いにしたいと考えます。過去のすべての報告集の論文について、著作権者（論文を執筆された故人の相続人）を探し出して利用許諾に関する同意を頂くことは困難ですので、一定期間の権利者捜索の努力をしたうえで、著作権者が見つからない場合も論文を情報学広場に掲載させていただきたいと思います。その後、著作権者が発見され、情報学広場への掲載の継続に同意が得られなかった場合には、当該論文については、掲載を停止致します。

この措置にご意見のある方は、プログラミング・シンポジウムの辻尚史運営委員長 (tsuji@math.s.chiba-u.ac.jp) までお申し出ください。

加えて、著作権者について情報をお持ちの方は事務局まで情報をお寄せくださいますようお願い申し上げます。

期間：2020 年 12 月 18 日 〜 2021 年 3 月 19 日
掲載日：2020 年 12 月 18 日

プログラミング・シンポジウム委員会

情報処理学会著作権規程
https://www.ipsj.or.jp/copyright/ronbun/copyright.html