Algol60

プログラム言語論第六回

3.1. 歴史と動機

- 国際的な言語が求められていた.
- Algol-58, そして Algol-60 が設計された.
- 形式的文法を用いて構文規則を記述した.
- "report" は簡潔さ(短く明確)の典型.

形式的文法を用いたのは画期的！

歴史

- (一企業に依存しない)国際的かつハードウェア独立な言語が必要とされていた
  - 提案: 1957
- ALGOL-58
  - 当初の名称: IAL – International Algebraic Language
  - 第一版はチューリッヒで設計された (8 日で)
  - 標準化には至らなかった

ALGOL-60

- "Algol-60 Report" が 1960年5月
- Algol-58 とは大きく異なる
- 誤り(あったのだ)の訂正が "Revised Report" で

"Revised Report on the Algorithmic Language ALGOL 60" を検索してみよ

設計:
構造の構成

3.2. 構造の構成

歴史的な出来事！

- Algol プログラムは階層的な構造を持つ.
- 各要素は宣言的または命令的.
  - 宣言: 変数, 手続き, スイッチ.
    - 変数(の型): integer, real, Boolean.
    - 手続きは型がある場合(関数)とない場合(サブルーチン)がある.
    - スイッチは計算型GOTOとして使用.

計算型GOTO: FORTRAN
例: GO TO (100,200,300), I

階層的構造

```
for i := 1 step 1 until N do
        sum := sum + Data[i];
begin
        integer N;
        read int (N);
        begin
                real array Data[1:N];
                integer i;
                sum :=  0;
                for i := 1 step 1 until N do
                        begin
                        end
                …
        end
end
```


階層的構造 (続)

人間にとっての分かりやすさが不十分

- 下記の形も許される

```
if N > 0 then
  for i := 1 step 1 until N do
    sum := sum + Data[i];
```


構成要素

- FORTRAN と同様に
  - 宣言(型要素)
  - 命令(型要素)

宣言型要素

- 変数
  - Integer
  - Real
  - Boolean
  - 配列は静的も動的も可. 添え字の下限は0または1である必要なし！.
- 手続き
  - 型付き (関数の場合. 戻り値の型)
  - 型なし(サブルーチンには型は不要だから)
- スイッチ

命令型要素

- "命令型"要素としては、計算と実行制御に関するもの(入出力に関するものは考えない).
- 計算
  - 代入 (:=)
- 実行制御
  - goto
  - if-then-else
  - for ループ

コンパイル時か実行時か

- Algol のデータ構造は、FORTRANに比べ、実際に作成される時期が, 一般的に, 遅い.
  - データ領域には, 作成・開放が実行時になされるものがある.
    - E.g. 動的配列, 再帰手続き
    - 識別名が実際のメモリアドレスに結び付けられるのは, 実行時,
    - 一方, 型に結びつけられるのは, コンパイル時.
- スタックは、実行時に用いられるデータ構造の中心的役割.

設計:
命名の構造

3.3. 命名の構造

- 基本的な"命名構造"は、名前を対象に、宣言によって、結び付けることである.
- 基本的には、構築道具は、ブロック.
- Regularity: 一つの文が現れてよいところには、複数個の文をグループにして書くことができる.

ブロック

- ブロックを用いると名前の有効範囲をネストすることができる.

begin declarations; statements **end**

- ブロックを用いると、大きなプログラムの作成が単純化される.
- Algol における共有データ構造は、一度定義されればよく、したがって、不整合性は生じ得ない(「間違いを生じさせない工夫」原則)。Fortran のCOMMONと対比するとその意味がわかる

重要: Impossible Error Principle

誤ることが不可能なようにすることの方が、
誤りが発生してからそれを検出するより、
望ましい.

名前の有効範囲(scope)

- FORTRAN
  - 大域的: サブルーチン名
  - 局所的: 変数, COMMON
- ALGOL
  - 有効範囲はネストできる.
  - 自分を含む有効範囲にはアクセス可能
  - 深刻な混乱を招くことがある

有効範囲のネスト(nested scopes)

```
real x,y;
        begin
                real x;
                begin
                        real y
                end
        end
        begin
                real z;
                x := 3;
                y := 4
        end
```

ブロック（block）

- 構造を簡潔にする
  - 共有構造の抽象化を後押しする
  - ブロック間のデータ共有を可能としている
  - 宣言を繰り返す必要なし, FORTRAN の COMMON は繰り返す必要がある
- しかし、どんなアクセスも許している (まだ情報隠蔽の考え方はない)

静的 or 動的な有効範囲
（static/dynamic scoping）

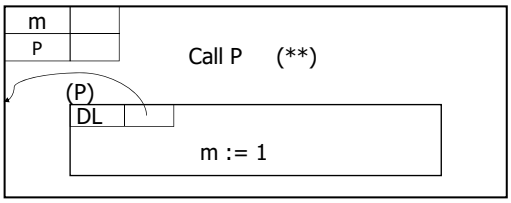
- Dynamic scoping: 文や式の意味は、時間とともに変化する計算の動的な構造（ってわからないよね）によって決まる.
- Static scoping: 文や式の意味は、計算の動的な変化に依存せずに、その静的な構造で決まる（これも難しいが、「プログラムの字面」で決まるということ）.
  - 「プログラムの字面では一意には決まらない意味」というものを定義することはできるのです。


```
a: begin integer m;
     procedure P;
         m := 1;
     b: begin integer m;
             P                    (*)
        end;
       P                          (**)
     end
```

P の呼び出し
外側のブロックから

```
a: begin integer m;
     procedure p;
         m := 1;
     b: begin integer m;
             P                    (*)
        end;
       P                          (**)
     end
```

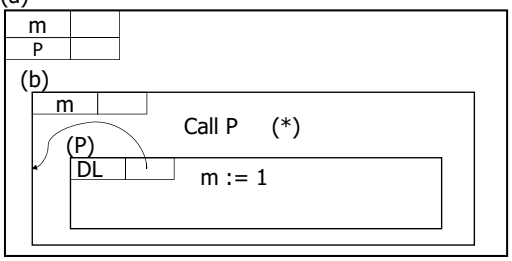
(a)



P の呼び出し
内側のブロックから

```
a: begin integer m;
     procedure p;
         m := 1;
     b: begin integer m;
             P                    (*)
        end;
       P                          (**)
     end
```

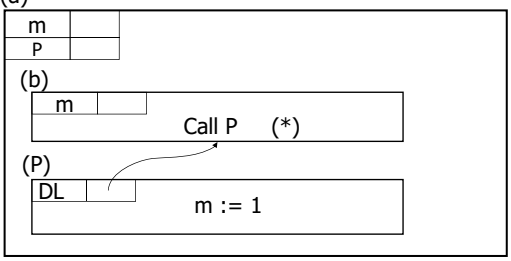
(a)



P の呼び出し
定義環境から

```
a: begin integer m;
     procedure p;
         m := 1;
     b: begin integer m;
             P                    (*)
        end;
       P                          (**)
     end
```

(a)



```
begin
  real procedure sum;
  begin real S, x; S:=0; x:=0;
     for  x:= x+0.01 while  x<=1  do
        S := S + f(x);
      sum := S/100
   end;
    …
end
…
begin
   real procedure f(x);
      value x; real x;
       f := x^2 + 1;
   sumf := sum
end
```

Static/Dynamic Scoping

- Static Scoping
  - 有効範囲はコンパイル時に決まる
  - 手続きが実行される環境は、定義された時点で(つまりプログラムの字面で)決まる
  - ALGOL 等で採用
- Dynamic Scoping
  - 有効範囲は実行時に決まる
  - 手続きが実行される環境は、呼び出された時点で(つまり実行中に)決まる.
  - Lisp が典型

Dynamic Scoping

- 最近では、一般には用いられない
- Lisp でさえ、新しい版では使わないものがある.

ブロック

- 通常は、メモリは、すべてスタック上にとる
- プログラム上では、BEGIN…END で区切る
- ブロックに実行が入ると, そのブロックの局所変数の領域がスタック上に取られる
- ブロックから実行が出ると, 局所変数はスタックから廃棄される
- ブロックはネストするか、排他的かのいずれか
- つまり、部分的に重なることはない

- ブロックを用いて局所変数を使うと、メモリの効率的な利用が図られる.
  - FORTRANにおいては、例えば EQUIVALENCE 文を用いて、メモリ領域の効率的利用を図ったがその必要はなくなる.
- 設計の責任原則(responsible design principle):
  - ユーザに何が欲しいかを聞いてはいけない; 何を望んでいるかを発見すべきである.

不要になったメモリを解放する。
ブロックを用いると、スタックで実装でき

マーケティングで重要な点。永遠に真であろう

設計:
データ構造

3.4. データ構造

- 基本要素は、数学でいえば、スカラー.
  - Integers, real, Boolean

データの基本要素（primitives）

- スカラー
  - Integer
  - Real
  - Boolean
- 倍精度はなし (ハードウェア独立にしたい)
  - 可搬性がなくなることを避けたかった
- 複素数なし（あるのはFORTRANぐらいだが）
  - real を用いればプログラムできるため

- FORTRAN （当時）では次の制限があった
  - 最大 19 行: 継続行continuation cards
  - 最大 6 文字: 変数名
  - 最大 3 次元: 配列の次元

0-1-無限大原理
（zero-one-infinity principle）

- プログラミング言語の中で、合理的な数というのは, 0, 1, 無限大だけである.
  - ある対象（例えば、フォルダの個数）の個数は、0か（つまり許されないか）、1か、制限なしのいずれかであるべきである。

（Willem Louis van der Poel (1926年生) . オランダのコンピュータ科学者）

動的配列（dynamic arrays）

- スタックを用いると、動的配列が実現できる.
- Algol60の設計は、柔軟性と効率性のよいトレードオフを達成している
  - 動的配列はスタック上に単純に実装できる

効率的な実装には、ハードウェアのサポートが重要

- 添え字の下限は、defaultでは0
  - a[100:200] と指定すれば, 100から200となる
- 配列は動的に配置することができる

```
begin
   integer i, j;
   i:=-35;
   j:=68;
          begin
          integer array x[i:j]
          end
end
```

強い型付け

- Algol は強い型付けの言語である:
  - 意味のない演算を, 間違って, 行ってしまうことを予防する.
- 型変換と型融合が許される.

設計:
制御構造

3.5. 制御構造

画期的！

- 制御構造は, FORTRANより一般化された.
- 文のネストを考え出したことは重要な貢献.
- 複合文(compound statements)は階層的構造.
- この概念の延長に構造的プログラミングがある.
- 手続きは再帰的に定義できる.
  - Activation record 中に複数個のインスタンスが現れてよい、ということ.
- ----------

if 文

- if 式 then 文1 else 文2;
- 式の中に用いることができる
- C := if A > 0 then 1 else 2;
  - 等価:
    if A > 0 then C := 1
    else C := 2;
  - C 言語の条件式と等価

複合文(compound statements)

- begin-end ブロックは文となる
- begin-end ブロックは、文が現れてよいところならどこにおいてもよい
- これによって、FORTRAN IV の if 文の制約(if 文には一文しかかけない)が解消された

ちょっとした問題

```
for i := 1 step 1 until N do
      ReadReal(val);
      Data[i] := if val < 0 then –val else val;
for i:= 1 step ….
```

は、実は、次のようにしたかったのだ:

```
for i := 1 step 1 until N do
      begin
      ReadReal(val);
      Data[i] := if val < 0 then –val else val
      end;
for i:= 1 step ….
```

- **begin-end** という「括弧」
  - 複数個の文を一つの複合文に纏める
  - ブロックを（他から）区切る, 有効範囲（scope）のネストを定義する

手続き

- 本来的に、再帰的
- パラメータは、値渡しができる。名前渡しもできる（Algolとしてはこっちが売りであったが）

値渡し

```
Procedure switch(n);
        value n; integer n;
        n := 3;
        …
```

- これで FORTRAN での「定数が変わる」問題が避けられる
  - FORTRAN で試してみよう。今のFORTRANコンパイラは対処してくれるかな？

名前渡し

- 置換（substitution）に基づく

```
procedure Inc (n);
    value n; integer n;
    n := n + 1;
```

- このとき Inc(k) は何をするか?
  - 何もしないよね – 値渡しなら
- 名前渡しならどうなるか (これが default)

```
procedure Inc (n);
    integer n;
    n := n + 1;
```


名前渡し

- 参照渡し（call by reference）ではない.
- （先ほどの例であれば）手続き側の "n" を、呼び出し側の "k" で置き換える.

（いじわるな）例

```
procedure S (el, k);
    integer el, k;
    begin
        k := 2;
        el := 0
    end;

A[1] := A[2] := 1;
i := 1;
S (A[i], i);
```

(意地悪な)例 (続)

- あたかも次のように書かれているかのように実行する:

```
procedure S (A[i], i);
    integer A[i], i;
    begin
            i := 2;
            A[i] := 0
    end;
```

- もし, A[i] := 0 を期待していたなら、それは誤り
- 蓄蓄: 実装機構は thunk とよばれるものを使用.
  - 参考: http://en.wikipedia.org/wiki/Thunk

有名な活用例: Jensen's device

- $x = \Sigma_{i=1,n}\ V_i$ x := Sum(i,1,n,V[i])

```
real procedure Sum(k,l,u,ak);
  value l,u;  integer k,l,u; real ak;
  begin real S; S:=0;
     for k:=l step 1 until u do
          S:= S+ak;
      Sum:=S
  end;
```


- Sum(I,1,n,B[I]*C[I])

- この手続き Sum は非常に一般的である
- $x = \Sigma_{i=1,m}\ \Sigma_{j=1,n}\ A_{ij}$
- どうなる?

実装: ありうる方法

1. 実引数の「文字面」を手続きに渡す
   - コンパイルし, この文字面を, 当該引数が参照される度に実行する
2. 実引数をコンパイルして機械コードを作成し, 当該引数が参照されるすべての場所に、この機械コードをコピーする
   - 何回もコピーすることになる
3. →実引数のコンパイルしたコードのアドレスを渡す, これが ***thunk***

Thunks

- 引数のないサブルーチン
- 引数が参照されるごとに, 呼び出された側(手続き側)は当該thunkを実行する.
- Thunk実行の結果, 変数のアドレス, は呼び出された側(手続き側)に戻される(関数のように).

見えない, 引数のない, アドレスを返す, 関数

- x:=Sum(i,1,m,Sum(j,1,n,A[i,j]))

Sum (k,l,u,ak)

k → i
l = 1
u = m
ak → thunk:
Sum(j,1,n,A[i,j])

Sum (k,l,u,ak)

k → j
l = 1
u = n
ak → thunk:
A[i,j]

Thunk における変数の有効範囲

- 引数が引き起こした変数間の関係は、呼び出し側にも影響を与える.
- Sub(y) を Sub(x) で呼び出したとき、もし、Sub の中に x があると、わかりにくいことが起こる.
  (マクロのように単純なわけにはいかない)
  (読んで理解することが難しいプログラムとなる可能性がある)


```
int c; //global variable
…
Swap(int a,b) {
   int temp;
   temp:=a; a:=b;
   b:=temp; c:=c+1;
}

y(){
   int c,d;
   swap(c,d);
 }
```

名前渡し

- 強力である
- しかし、わかりにくい
- 実装にはコストがかかる (実行時の thunk の呼び出しもコスト)
- どちらを default にすべきか?

- 2変数の値を入れ替えるプログラムを書いてください.
- それは、どんな実引数に対しても正しく動きますか?
- その理由は?

プログラミング言語の概念モデル

- Don Norman[1] (認知心理学者): "A good conceptual model allows us to predict the effects of our actions."
  - 設計者のモデル – システム構造を反映する
  - システムイメージ – 設計者によって作られる; ユーザのモデルの基盤となる – マニュアルや図表も含む.
  - ユーザのモデル – システムイメージ、ユーザ個人の適性や個人が感じる心地よさに基づき、ユーザによって作られる.

[1] *Psychology of Everyday Things* (Basic Books, 1988)

ブロック脱出 goto

```
    begin
        begin
            goto exit;
        end
exit:
    end
```


ブロック脱出 goto

- 単なるジャンプではない！ことに注意
- あたかもブロックの最後のendを通り抜けたかのように、ブロックを終了すべし
  - 変数領域の開放
  - activation record の廃棄
- では、ブロックの中へ飛び込むことはできるか?

特徴の相互作用

Algol の可視性とgoto文とが合わさると
実行の効率性が低下するおそれあり

仮に 100 個の特徴があれば,
100^2 個の2特徴間の相互作用が
100^3 個の3特徴間の相互作用が
…

for-ループ

- 2個の基本形:
  - for *var* := *exp* step *exp′* until *exp″* do *stat*
  - for *var* := *exp* while *exp′* do *stat*
- また
  - for days := 31, 28, 31, 30, 31, *…,* 31 do *stat*
- 芳しくない？でももっとあるのです

更なる複雑さ

```
for i := 3, 7,
      11 step 1 until 16,
      i/2 while i ≥ 1,
      2 step i until 32
  do print (i);
```

- 3,7,11,12,13,14,15,16,8,4,2,1,2,1,2,4,8,16,32

原則からの逸脱

- for i := m step n until k do …
- ALGOL の仕様は〝 m, n と k はループの各繰り返しごとに再評価される”
- 再評価は各繰り返しに行わなければならない. たとえ値が変わらなくとも、また、それが定数であっても
- この実行コスト上昇はすべてのループにわたって発生する(コストの分散負担), たとえ m, n, k が定数であっても.

コストの局在化原則

ユーザーが払うコストは、
ユーザが使うものに
限るべきである;
コストの分散負担は
回避すべきである.

Switch 文

```
        begin
                switch marital status = single, married, divorced, widowed;
                …
                goto marital status[I]
single:         … handle single case
                        goto done:
married:                … handle married case
                        goto done:
divorced:               … handled divorced case
                        goto done:
widowed:                … handle widowed case
done:
        end;
```


ちょっと怪しい Switch

```
begin
        switch S = L, if i > 0 then M else N, Q;
        goto S[j];
end
```

注意: S の値は 3 個ある:
- L
- If i > 0 then M else N
- Q

- 引数は、値渡しができる.
- 値渡しは、配列に対しては、この上もないほどに非効率的になりうる.
- 名前渡しは、"置換" に基づく.
- 名前渡しは、強力である.
- 名前渡しは、危険でありコスト高である.

- 概念モデルがよいと、ユーザは助かる.
- ブロック脱出goto文は、コスト高になりうる.
- 特徴・機能の相互作用は、設計問題としては、かなり難しい.
- Algol の for-ループ は、非常に一般的.
- Algol の for-ループ は、飾り立てすぎ.
- Algol の switch は場合わけに対処するため.
- Algol の switch は、飾り立てすぎ.