E1　最近の汎用グラフィック言語

出羽　洋，山本欣子（日本情報処理開発センター）

はじめに

汎用グラフィック言語の機能として何を要求するかということは一口につくせない．それには，コンピュータ・グラフィックスの応用分野が余りに広いということ又そのソフトウェアー体系がまだ完全に確立されていないということが理由としてあげられるだろう．C. I. JOHNSON はハイレベルのグラフィック・ランゲッジとして要求される諸機能に関して次の5項を挙げている．

1. 割り込み処理機能
2. 表示用図形データ構造処理機能，図形入出力
3. 問題向けデータ構造処理機能
4. 図形生成・変換機能
5. 計算機間コミュニケーション機能

この中には，いわゆるグラフィック・オペレーティングシステムといわれるものがどこまでサポートするかによって考え方が変わってくるものもある．

ここでは1°，4°に焦点をしぼり，比較的最近のグラフィック言語といわれるものを中心にその問題点等にふれてみたい．

＜割り込み処理機能＞

割り込み処理は，グラフィックにおけるインタラクティヴ・プロセスを実現するための本質的な機能である．これは，具体的には，ファンクションキー，ライトペン等の各種の割り込み要因に対して，処理プログラムの動作をどう対応づけるかの方法の問題といえる．まず，割り込みの取扱い方法という点からこれを非同期処理，同期処理に分けて考えてゆくことにする．

1. 非同期処理

本来，インタラクティヴ・プロセスとは，「READ & REPLY」，「WRITE & WAIT」という形，つまり結果をみてActionを起すという形で進められてゆく性格のものである．

しかし，ある状態，例えばある種の割り込みモードで動作中に，一担それを中断して別の処理を要求するというような事態が発生した時これは同期処理系では扱えないものになる．このような処理を可能にするためには，非同期処理能力を備えていなければならない．

ところで，一般に非同期処理を扱う系では，これを処理するためのプログラムは特別な構造を備えている必要がある．即ち割り込まれる前に使用していたプログラムリソースが共有され

る可能性があるため，もとの状態に正しく復帰させるためには，処理プログラムはリエントラント構造を備えていなくてはならないという事である．

しかし一般に，PL／Ⅰを除く既存の言語はリエントラントなオブジェクトプログラムを作成する能力を持たないため，既存の言語をベースにした時この種の処理を実現することは仲々困難であり実際には余り考慮されていないようである．しかし，少なくともユーザが意識的にリエントラントなプログラムを作成した時（例えばアセンブラー等で）にはそれに対して非同期処理が認められるような考慮は加えておくべきであろう．

2. 同期処理

同期処理の方法に関しては，次の2つの処理形式が一般的である．

① テーブル記述形式

② プログラム・ステートの遷移という形で表現する方法

①は各種GSP（グラフィック・サブプログラム・パッケージ）で広く用いられている方法である．

即ち，あらかじめ割り込み要因と処理ルーティンの対応を与える制御表（テーブル）をサブルーチン・コールの形式で作成しておき，割り込みが発生した時この制御表をもとに処理ルーティンが呼び出されるという形式のものである．この時，割り込みの受けつけに関しては，テスト命令を実行したり，あるいはサブルーティン呼び出しの形で次に発生する割り込み要因に対して割り込みマスクを設定しておき，プログラムを割り込み待ちの状態におく等の方法がある．

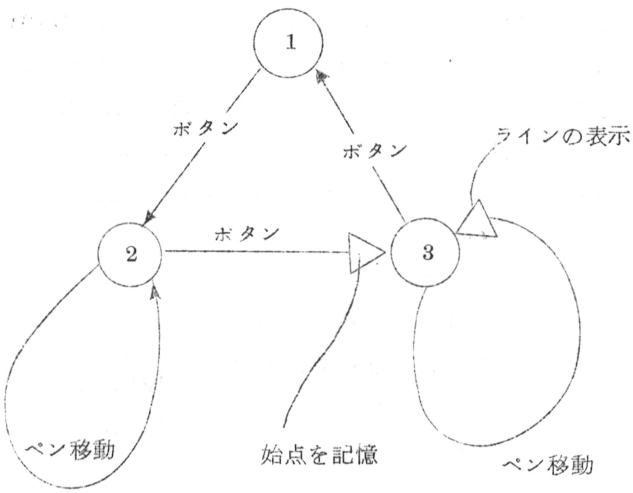
このように一連の割り込み処理過程を，割り込み制御表への登録・修正，割り込み要因の禁止・許容といった形で記述することは，割り込み処理の概念を持たない者にとっては理解しにくいものであるかも知れない．又プログラム動作モードの流れ（プログラムステートの流れ）というものを考えた時，それを把握するためにはプログラム全体に目を通す必要があり不便であろう．

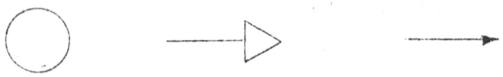
②では各動作モードにプログラムステートという概念を与え，より理解し易い方法をとっている．（これはW.M.NEWMAN等の考えに基づくもので，基本的にはインタラクティヴシステムを有限の状態を持つオートマトンと考えAction，Reactionをそれぞれオートマトンへの入出力として扱おうとする考えである） 即ち，各プログラム動作モードにプログラムステートを対応させ，各プログラムステートの定義の中でそのステートで可能な割り込み要因とその処理ルーティンの対応をはかり，かつ処理ルーティンの終りで次のステートへの移行を記述するという形式である．

この種の扱いをしている言語の例として，DIAL，AIDS 等がある．

例えば，Fig.1 のステートダイヤグラムで示される過程（ラバーバンドラインの表示）はDIALでは次のように表現されるであろう．

```
    ⋮
ENTER 1 ;
DURING 1 DO                                          ┐
     ON BUTTON DO BEGIN                              ├ ステート１の定義
                    ENTER 2 ;                        │
                    END ;                            ┘
DURING 2 DO                                          ┐
     ON PEN      DO BEGIN                            │
              …ライトペントラックの過程…             │
                    ENTER 2 ;                        ├ ステート２の定義
                    END ;                            │
     ON BUTTON DO BEGIN                              │
              …始点の記憶                            │
                 ……                                  │
                    ENTER 3 ;                        │
                    END ;                            ┘
DURING 3 DO                                          ┐
     ……                                              ├ ステート３の定義
                                                     ┘
```





State　　Reaction　　Action

Fig. 1

このように，この方法は，割り込み処理に慣れない者にとってもプログラムステート，アクション，リアクションという概念をつかめば容易にその過程を表現できること，又プログラム動作モードの流れのつかみやすさという点に於ても優れた方法であるといえよう．
AIDSも表現形式は異なるが基本的な考え方は同じである．

＜図形生成・変換機能＞

一般に複雑な図形を表現する時，これを構造化して扱うのが便利である．

この時，構造化の対象となるプリミティヴな図形が考えられるが，これを以後，イメージと呼ぶことにする．

図形生成・変換機能は，結局イメージの記述性，操作性という問題でとらえてゆけばよいだろう．これはイメージがどのように内部構造化されているかという事とも密接に関係している．まず構造化の方法としてこれを2つのタイプ ディスプレイ・プロセデュア型とディスプレイ・データ・ストラクチャー型に分け各々に特徴した問題についてふれてみる．

1. ディスプレイプロセデュア型

イメージの構造を，内部的にプログラムのつながりとして表現する方法である．

すなわち，イメージ間の関係はコンパイル時にすでに決定されており，その構造は図形データを生成するプログラムの階層構造として内部表現されている．表示データは，出力命令によりこれらの一部が実行されることにより作成される．

このカテゴリーに属するものとして，低レベルのものではGSP，進んだものではDIALがその例である．GSPではあらかじめ VECTOR，POSIT 等の図形データ編集のためのモジュールや，論理的な図形のまとまりを指定するためのモジュール ELMT，ENDELM が準備されており，これらのモジュールをCALL形式で呼び出すことにより図形記述を行なう事になる．そして実行時にこれらのモジュールがアーギュメントリストをデータにして次々と，表示データを作成してゆく．

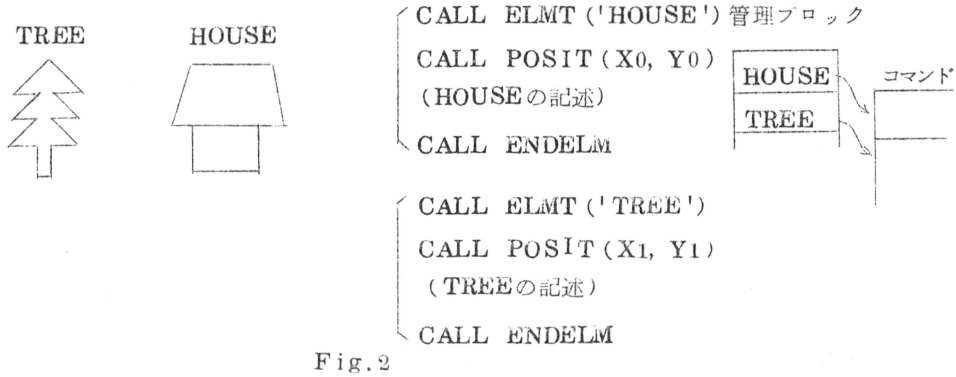


Fig.2

この形式は，実行時に，直接表示データを生成するから処理速度は速い．しかし図形データをアーギュメントリストで与えるため煩雑になり，余り記述性が良いとは云えないだろう．（又ディスプレーファイル上での構造化は，はかるがここでいうイメージの構造化機能は持たない．）

ところで，一般に図形を記述するのに基本的な図形を定義しておき，それを引用しながら，次々に複雑な図形を組み立てられたら便利であろう．DIALは，このような要求をよく満たしている．

DIALに於て，構造化の対象となる一つの図形のまとまり（イメージ）は，ディスプロセデュアと呼ばれるプロデュアによって定義される．

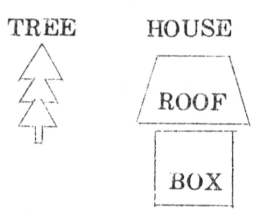
ディスプレー・プロセデュアは内部的には表示データを作り出すプログラムのまとまりとして表現されている．即ちプログラム上でイメージの構造化をはかることは，プログラムの間のつながりを定義していることに等しい．ディスプレー・プロセデュア内で図形データを定義するためのステートメントとしては次のものが用意されている．

MOVE X, Y	相対的な位置づけ
MOVE TO X, Y	絶対的な位置づけ
LINE X, Y	直線記述（相対）
LINE TO X, Y	直線記述（絶対）
DISPLAY '～'	TEXTの記述

イメージの表現はこれらのステートメントと他のディスプレー・プロセデュアの引用によって一つのディスプレー・プロセデュアを完成することによって為される．

例えば，Fig.2 で表示される例は次のように表現される．

（例）Fig.2




```
PICTURE     PICTURE←'HOUSE atX1, Y1, TREE atX2, Y2;'
HOUSE       ←'ROOF atX3, Y3 BOX atX4, Y4;'
TREE        ←'MOVE TO X1, Y1 LINE XN, YN …
              …TREEの記述;'
ROOF        ←'ROOFの記述 …;'
BOX         ←'BOXの記述 ……;'
```

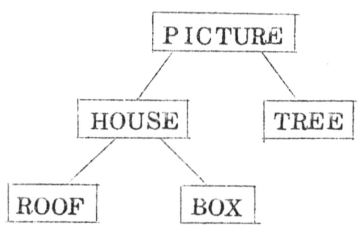


Fig. 3

Fig.3 はこれらのステートメントにより作成されるプロセデュアの階層関係を示している．この中の任意のディスプレー・プロセデュアを指定して表示命令（WINDOW）が実行されると関連したプロセデュアが次々と実行され，表示データが作成されてゆくことになる．

以下にDIALにおける図形の扱いを，イメージの操作性という点から整理すると次の様になる．

1. ディスプレー・プロセデュアの構造はプログラム作成時にスタティックに決定される．
2. ディスプレー・プロセデュアを引用する時その位置情報，スケール情報を与える事が出来る．
3. ディスプレー・プロセデュアにアーギュメントを与える事が出来る．
4. ディスプレー・プロセデュア内に判断文，繰返し文等のステートメントを書く事を認めている．

1.は実行時に動的に図形構造を変える事の出来ない事を意味する．これは構造化の方法から来る大きな問題である．

3.，4.は，この問題（イメージの動的な操作性）に対してある程度その機能を補ってくれるだろう．

2.は，定義した図形を引用する時にどの程度までに，それに操作を加えられるかという事で，サイズや位置情報は自由に変えることが出来る事を示している．しかし回転等の操作は加えられない．これもやはり構造化の方法からくる問題である．

2. ディスプレーデータストラクチャー型

これはより一般的な方法，即ちイメージの構造をデータ構造として表現する方法である．図形記述ステートメントは実際にはデータストラクチャーの記述であり，実行時に内部にデータ・ストラクチャーが作成される．即ちイメージの実体はデータ・ストラクチャーとして在ることになる．そして実際の表示データは，表示命令が出された時にデータ・ストラクチャーを対象として作成されることになる．

この方法は，インタプリティヴに表示データが作成されるという点で，処理速度に問題を残すが，動的な図形構造の操作，イメージの扱いに関しては1の方法よりもはるかに自由度の高いものになる．この種の扱いをしている言語として例えばGRIN Ⅱ（GRAPHIC Ⅱシステムで使用されている）等があげられる．（この形式は2プロセッサー方式でホストプロセッサーに置く図形構造として適している．

GRIN Ⅱで扱うデータ構造はリーフ，ブランチ，ノードと呼ばれる，3つの基本的な要素から構成され，各々は次の情報を蓄えている．

○リーフ・ブロック……実際の図形データを蓄える．

○ブランチ・ブロック…図形変換情報（位置情報，スケール情報，表示条件）
割り込み制御のための情報

○ノード・ブロック……ノードの識別名

ここで，イメージに対応するものはリーフである．又ブランチはイメージを構造化する時に必要な情報であると考えられる．Fig.4はFig.2の例をGRIN Ⅱのデータ構造として表現したものである．

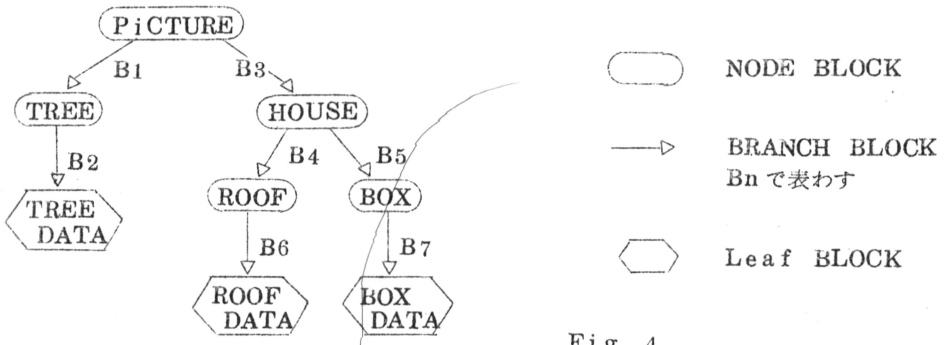


Fig. 4

図形記述は，下に示したステートメントを用いて為されるが，図形記述をストラクチャー記述という形で行なうため大変表現しにくいものになっている．

LEAF …… リーフブロックの定義

NODE …… ノードブロックの定義

BRANCH …… ブランチブロックの定義とノード間の結合

DETACH …… ブランチを取り除く

ATTACH …… ブランチをノードに付加する．

GRIN Ⅱ における記述例

```
LEAF  TREED
VECTOR ( DXY1 , DXY2 …  )  ( TREE データの記述)
LEAF  ROOFD
VECTOR ( …                 )  ( ROOF データの記述)
……
NODE  TREE ( ( B1 , , , TREED) ) ( リーフ , ブランチ , ノードの結合)
……
NODE  . PICTURE                    (ノードの定義)
BRANCH  B1 , , , , PICTURE         (ノードブランチの結合)
……
```

又 GRIN Ⅱ ではブランチ・ブロックにプロブレム・データを持たせることにより，図形の構造化と問題向け構造化を同時にはかることを試みている．

このように一つの構造の中に，図形情報と問題解析のための情報を表現しようとする行き方は，データ構造の利用度の高さという点から見れば確かに優れた方法であると云えるが，結局それを記述する段階で，図形構造と問題解析のための構造の両面のつじつまをうまく考えながら記述してゆかなくてはならず煩雑さをまねくことになるだろう．又，両側面をうまくみたしてくれるような構造化の方法があるという保証も無い．

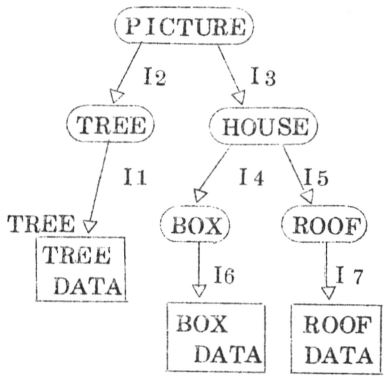
（このような事情から，最近では図形データ構造は専用の構造とし，いわゆる問題解析のための汎用データ構造と分離して扱うべきだという方向が強い．）

AIDSもGRIN Ⅱ とほぼ同じ形式の構造を持つ．

GRIN Ⅱでノード，ブランチ，リーフと呼ばれたものはAIDSではそれぞれセット，インスタンス，イメージと呼ばれる．（セットが構造化の対象となる図形を表わす）両者の相異は，AIDSでは問題向け構造は備えていないこと，又処理速度をあげるため データ・ストラクチャーがディスプレイプロセッサーにとって実行可能な形式で作られていることである（その反面イメージの操作性を犠牲にしている）．

Fig.5 はAIDSによる図形記述の例である．この過程は次のように要約されるであろう．

1. イメージの定義 … 実際の図形データを定義する．（INSERE image）
2. インスタンスの定義 … イメージ又はセットを引用する時に必要な位置情報を与えておく．
   （POSITION instance）
3. インスタンスとイメージ又はインスタンスとセットの結合をはかる．
   （instance DEFINES image）
4. セットの構造化（SET CONTAINS）



INSERT IMAGE TREED:LINE FROMXY
⋮ TOX′Y′……
⋮ TREEのデータ記述
⋮

POSITION INSTANCE I1 AT
⋮

SET PICTURE CONTAINS INSTANCE I2, I3
SET TREE CONTAINS INTANCE I1
⋮

——▷ インスタンス（Inで表わす）

セット

イメージ

Fig. 5

これもやはり，図形を記述をする際に，構造をかなり意識する必要があり余り記述性が良いとは云えない．

ここで注意したいことは，図形を構造化して扱えるということは確かに便利であるが，先の例に見るように意図する図形を記述するのに，それを図形データ構造との対応という形でとらえなおし，実際には図形データ構造の記述という形で表現することになると，かえって煩雑さが増し記述性を悪くすることになるということである．

構造化は内部的な表現形式であり，使用者にそれを意識させるべきでない．本来の，図形構成という考えだけあれば，自由に記述することが出来るというのが望ましいであろう．
GPL／I ではこうした事情をよく反映している．即ち構造化は内部的な表現手段であり，ユーザーからは見えないものでなくてはならないという点を強調している．

ここで，その機能の概略についてふれてみよう．

GPL／I は PL／I をベースにして作られており，従来の PL／I の機能に，グラフィック・データを扱う機能，グラフィック入出力機能，割り込み処理機能等を追加した形をとっている．GPL／I で扱う基本的な図形データとしてイメージとベクトルがある．イメージは操作対象となる図形を表わしベクトルはイメージの実体を定義するイメージ・データと考えられる．

ベクトルは位置ベクトルの概念によくあてはまり，２つ又は３つのコンポネントで定義され，各々２D，３Dの位置情報を表わす．例えば（１，５），（６，９）を結ぶ線分はイメージLとして次のように定義される．

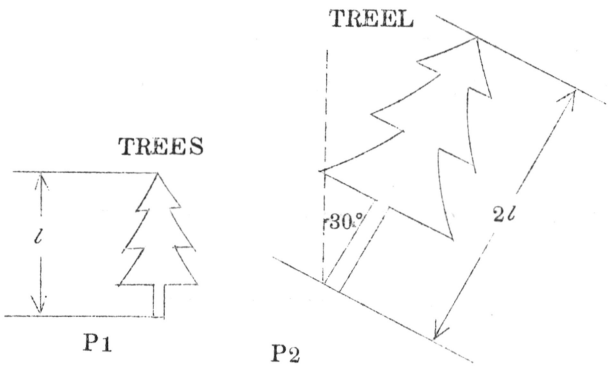
L＝（1X＋5Y）|||（6X＋9Y）

（X，Yはベクトルの成分を表わし |||はコネクションを表わすグラフィック・オペレーターである．）

ベクトルの他にイメージ・データを定義するものに TEXT，FUNCTION があるが，余り細部にはふれない．これらのグラフィック・データ，ベクトル，イメージに対し適用されるグラフィック・オペレーターには次のようなものがある．

＋＞	inclusion	を表わす
|||	connection	を表わす
@	positionning	を表わす
＜＞	scaling	を表わす
＊＞	rotation	を表わす

例えばFig.6で示されるTREESがイメージとして定義されている時TREELはこれを用い



Fig．7

て次のように表現されるであろう．

TREEL＝（（TREES<>（2X+2Y））*>（−30Z））@P2

このようにして，任意の図形は，グラフィック・データ，グラフィック・オペレーターのエクスプレッションとして自由に表現する事が出来る．

例えばFig.2の例をイメージ，エクスプレッションとして表現すれば次のようになるだろう．

HOUSE＝ROOF @P1 +> BOX @P2

PICTURE＝TREE @P3 +> HOUSE @P4

（P1～P4はベクトルで各イメージの位置を与えている）

この例からも分るように，ここでは，図形記述に際して先の例に見たような，構造化という意識は必要としない．つまり，より観念的なレベルで図形表現が出来るといえる．

GPL／Iはこの他にも色々便利な機能を備えている．

例えばイメージには各種のアトリビュートを与えることが出来る．これにはスタティックに決められるものとダイナミックに変更出来るものとがあるが，主としてイメージの表示条件に関係したものが多い（例えばDASHED属性を持ったイメージは表示されると破線で描かれる）中でも興味深いものにACTIVE属性がある．ACTIVE属性を持ったイメージが表示されている時プログラム上でそのイメージに操作が加えられるとそれが自動的に画面に反映されるというものである．

その他，イメージに適用されるFUNCTIONも各種あり主として，他のイメージのコピー，いくつかのイメージのグルーピング（論理的なまとまりにする），割り込み情報を調べる等の機能を果している．

又，3Dベクタルを使用して3Dイメージを定義する事も可能になっているが，完全に三次元図形を定義する機能を持たないためこれを，ワイヤクレームとして扱うにとどまっている．

あとがき

以上，いくつかの言語を例に，図形記述，操作という面から各々の特徴，問題点等にふれてきた．

しかしここで述べたことが，そのままその言語の評価にはあてはまらない．それはそれぞれの言語を開発した時の状況があり，ねらいとするところがそれぞれに異なるからである．

参考文献

1. C. I. JOHNSON: Principles of interactive systems IBM SYS. J. Vol. 7 NO.3 & 4, 1968
2. W. M. NEWMAN: A system for interactive graphical programming SJCC 1968
3. W. M. NEWMAN: An experimental display programming language

for the PDP-10 Computer, Information Research Laboratory University of UTAH 1969

4. CARL CHRISTENSEN: Multi-function graphics for a large computer system FJCC 1967

5. T. R. STACK & T. WALKER: AIDS-Advanced interactive display system SJCC 1971

6. DAVID N. SMITH: GPL/I-A PL/I extension for computer graphics SJCC 1971

7. その他

本 PDF ファイルは 1972 年発行の「第 13 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