

13. イオタ語の言語処理およびプログラム 作成支援システム

京 都 大 学
数理解析研究所

湯 浅 太 一, 中 島 玲 二
Taiichi Yuasa, Reiji Nakajima

§1. はじめに

ソフトウェアの信頼性向上やプログラム検証の困難さの軽減に、データ抽象化 (data abstraction) や手続き抽象化に基づく、階層的でモジュラーなプログラミングが有効であることは広く認められている。またこの方法によるプログラム開発に適した環境を与える研究が行なわれている [例えば 6]。

一般にこれまでの研究は主にプログラム言語の設計が中心であったが、プログラム言語はより良い環境を与えるための一要素にすぎず、言語の目的とする方法論が有効に働くためには、その言語の埋め込まれる環境が重要である。特に階層的でモジュラーなプログラミングは、個々のモジュールと同時に、モジュール間の相互関係を重視するので、プログラム言語やプログラミング・システムを含めた総合的な考察が必要である。このような目的から、イオタ・システムとよばれる階層的でモジュラーなプログラミングを支援するための総合的なシステムが開発されてきた。本論では、このシステムを概観し、その考え方と方法を述べる。イオタ語とその数学的基礎、検証方法などについて詳しくは [9, 10] を見られたい。

§2. 言語

イオタ・システムはプログラムと形式的仕様 (以下 '仕様' と略す) の記述のための言語、イオタ語を基礎としている。イオタ語で書かれるモジュールは、そのモジュールで定義される関数の宣言や仕様などを与える抽象的部分 (抽象部分, abstract part) とそのプログラム実現を記述する部分 (実現部分, implementation part) が明確に区別して与えられる。あるモジュール M が他のモジュールの上に作成される時、M はこれらのモジュールの抽象部分のみに依存し、それらの実現部分には全く依存しない。(以下モジュール間の '依存関係' はこの意味で用いる。)

イオタ語は支援システムに埋め込まれることを仮定して設計されており、言語の機能をこえると思われるものはシステムにまかせることによって、言語自体は簡潔で統一のとれたものとなっている。例えば、各データ型の等号 (equality) と代入 (assignment) の実行が禁止されていないかどうかの管理がそうである。またインラインコードなどによる最適化の指定もシステムへのコマンドによって行なわれる。イオタ語における仕様は、イオタ・システムに於いて、プログラムの開発と変更のためのツールとしての機能をもつ。モジュールの実現は、そのモジュールを procedural に記述したものであり、仕様は同じモジュールの declarative で抽象的な記述である。両者は互いに補間的な役割をもち、その記述を平行して行ない検証することにより、プログラムの信頼性が保証しやすいと考えられる。(階層的なプログラム開発においては、階層の下位に属するモジュールは、仕様も実現も多分に上位モジュールの実現のための便宜的存在であることが多いことに注意 [12].) このような仕様の利用法は、イオタ語の仕様が部分的 (partial) 仕様であることに基づいている [11]。

§3. 各モジュールを独立に処理すること

階層的でモジュラーなプログラミングは、元来主としてプログラムの検証の困難を克服するために提案されたものであり、検証がモジュールごとに行なわれるのは当然であるが、プログラムのモジュール構造を有効に利用するためにはプログラムの入力、変更、デバッグ、コンパイルなどもモジュール単位に行なえることが望ましい。

モジュールごとのコンパイルには例えば次のような利点がある。第1に、あるモジュールの実現に変更があった場合、既にコンパイルされた他のモジュールは影響をうけないのでコンパイルしなおす必要がない。実行時間の短縮のためには、最終的に完成した一連のモジュールを一括してコンパイルすることも望ましいが（種々の最適化がこれによって可能となる。）、プログラムの維持管理のためにはモジュールごとにコンパイルされたコードをプログラムデータベースに保つ必要がある。さらに、モジュールごとの構文解析や動的デバッグが有効に行なえる。

イオタ語によって書かれたモジュールは `synt{ 9 }` の概念を用いて `type` parameter 化が可能であり、この目的のためには特殊なテクニックが必要である。簡単のため trivial な例をあげる。いま2つの引数 x と y ととり、その大きい方をかえす関数 'max' を作りたいたい。max の引数の属するデータ型は、元 (object) の間に全順序関係 ' $\leq$ ' が定義されていれば何でもかまわない。この条件を記述するために `synt 'order'` が作成される。（図I参照。ここで '@' はこのモジュールで定義される `synt` 即ち `order` をあらわす。'AS' によって関数名の関数記号による省略が導入される。つまり `less(x, y)` と書くかわりに $x \leq y$ と書くことを意味する。）この `synt` の上に、max を定義するモジュール `sort(p: order)` がつくられる。（図II参照）ここで `p` はタイプパラメーターで、`order` の条件をみたす `type` に束縛される。REALIZATION 部には `↓max` (max の実現関数) の body が与えられている。IF 文の条件式 ' $x \leq y$ ' に注目されたい。`p` がある `type T` に束縛された時は、' $\leq$ ' は `T` で定義された関数をあらわす。例えば `p` が `type integer` に束縛されれば、`integer` の ' $\leq$ ' がよばれるであろう。このように、この条件式でよばれる関数は `p` がある `type` に束縛されてはじめて決定される。ではどのようにして `sort(p: order)` を独立して（即ち `p` の束縛される `type` に依存しないで）コンパイルできるだろうか？ この問に対する1つの解答は `CLU[1]` で与えられているが、イオタ語では `synt` を利用することによってより効率的な方法が開発されている。まずあるモジュール `M` で定義される関数の body で max がよばれ、このとき `p` はある `type T` に束縛されるとしよう。`M` をコンパイルする時に、`order` で定義された関数に対応する `T` の関数のテーブルを、それらの関数が `order` の INTERFACE 部にあらわれる順序に従って作成し（`order` の場合関数が $\leq$ しかないので単純だが一般の `synt` は複数個含む）、このテーブルを実行時に max におくる。`↓max` の IF 文の条件式では、やはり `order` の INTERFACE 部を調べることによって、このテーブルから実際によばれる関数を取り出してよぶようにコードが作成されている。

一般の場合は、タイプパラメーター機能に伴う問題はこの例のように単純ではなく、本質的な困難さを含む。一般的な問題点とその解決の方法については [13] に詳しく述べられているので参照されたい。

INTERFACE SYPE order

FN less: (a,a) -> a AS a=<a

END INTERFACE

SPECIFICATION SYPE order

VAR x,y,z:a

AXIOM 1. $x \leq y \vee y \leq x$
2. $(x \leq y \ \& \ y \leq z) \Rightarrow x \leq z$
3. $(x \leq y \ \& \ y \leq x) \Rightarrow x = y$

END SPECIFICATION

図 I

INTERFACE PROCEDURE sort(p:order)

FN max: (p,p) -> p

END INTERFACE

REALIZATION PROCEDURE sort(p:order)

FN ^max (x,y:p) RETURN (z:p)
IF $x \leq y$ THEN $z := y$
ELSE $z := x$
ENDIF

ENDFN

END REALIZATION

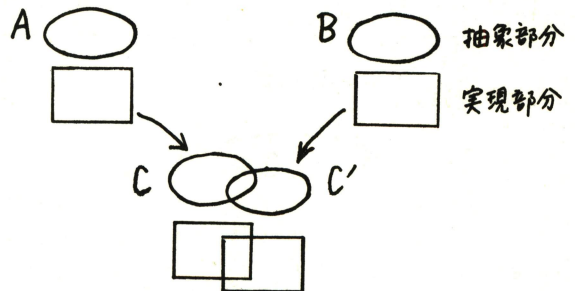
(ここで ^max は DEC 上の $\downarrow \max$ の表示, $\leq$ は $\leq$, $\vee$ は OR, $\&$ は AND, $\Rightarrow$ は "ならば" である。)

図 II

イオタ語のコンパイルに関するもう一つの重要な問題は、代入文や関数呼び出しなどの際の値の受け渡しである。一般に(抽象)データ型をユーザーが定義できる言語においては、arrayやrecordなどの組合せにより、個々の値が物理的に大きく得る。例えばイオタ語では代入文 $x := y$ を実行する場合、意味論上は y の値を複製(copying)することになっているが、実際のコードではできる限り意味論上の正しさを保ちながら、不必要な複製を制限するように最適化されている。これに関しては[14]が詳しい。

§4. モジュールとモジュール間の関係の管理

一度作成されたモジュールは絶えず修正の可能性を含んでいる。例えば右図において、モジュールAは検証までCの仕様を用いて完了しているとする。このときCの上にBを書きたいが、Bの要求がAとやや異なる場合、CをかえることによってAに費した努力がそなわれることになりかねない。このためCを若干変更したものを新しく作り、Cとともに並

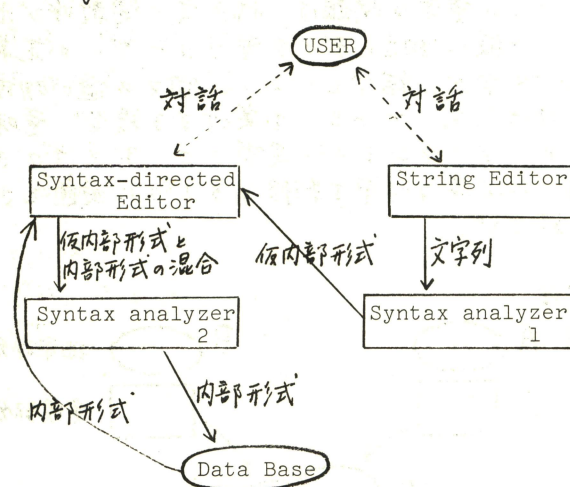


存させると都合がよい。一般に同じモジュールの多くのバリエーションを仮想的に維持することが便利であり、イオタ・システムにとり入れられている[11]。(前述の、イオタにおける仕様のあり方と関係していることに注意されたい。)

このような目的に加え、モジュール間の関係が circular になることを防止するなどのために、システムは常にモジュール間の関係を管理する必要がある。この管理はすでに完成したモジュールについてのみでなく、未完成のものも含む。

§5. モジュールの入力と変更

システムがプログラムの作成を支援するためにはユーザーとの対話が非常に望ましい。対話を効果的に行なうためには、システム全体に共通したプログラムの内部表現を持ち、各サブシステムがこれを直接扱える必要がある。LISPのように、内部表現とプログラムテキストの対応が明らかな場合はこれだけ容易だが、ALGOL型の言語では、内部表現への変換には構文解析を必要とし、この目的のためにはプログラムの入力と変更に特別な工夫が必要である。特にイオタ語の場合、モジュールの構文解析は、それが依存する他のモジュールの抽象部分にある情報を用いて処理することが必要であり、かなりの時間を要する。例えば、関数はその関数を定義するモジュールの名前と関数名によって一意に決定されるが、プログラムの中の式(expression)では、関数名だけ、あるいは infix 又は prefix の関数記号による省略(AS abbreviation)も可能である。同一の関数名や関数記号が二つ以上のモジュールで導入されている場合は、式全体のタイプチェックによって関数を識別する必要がある。従って、プログラムの変更の際には、一度作成された内部表現を有効に利用し、重複した処理を避けることが望ましい。このような理由から、従来の scanner や parser の単純な組合せによる schema では不十分で、次のような Syntax analyzer, Syntax-directed editor, String editor の有機的な結合が効果的である。



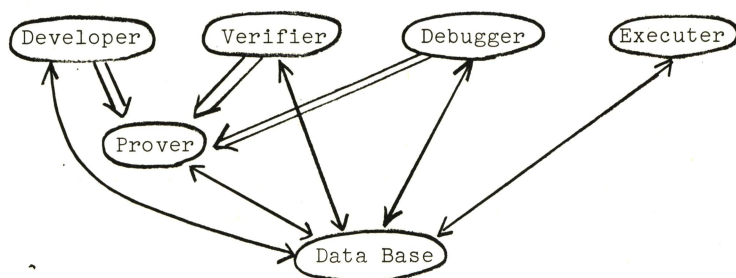
新しいソースプログラムは String editor を介して Syntax analyzer 1 に与えられ、大まかな構文木(仮内部形式)が作成される。ここでエラーが検出されれば自動的に再度 String editor がよばれ、(エラーの発見された部分にポインターがセットされて)エラーの局所的な修正が行なわれる。String editor には reserved word やユーザーの与えた identifier を自動的に展開する機能やフリタイプリントなどが備わっている。

Syntax analyzer 1 で作成された仮内部形式は Syntax analyzer 2 によって、データ型の分析、式にあらわれる関数の識別などが行なわれ、内部形式に変換されると同時にモジュール間の関係に関する情報がとり出される。

ここでエラーがあれば, Syntax-directed editorを通じて修正される。この際の入力は再度 String editor, Syntax analyzer 1 を通して行われ, 修正部分が新たな仮内部形式におきかえられる。この結果, Syntax analyzer 2 がすでに処理した内部形式の中に仮内部形式が混在したものができる。これを再度 Syntax analyzer 2 が処理するのであるが、仮内部形式の部分のみを処理し、すでに処理済みの部分はそのまま保存するように設計されている。これはイオタにおける構文解析が時間がかかることから大いに有効である。

§6. イオタ・システムの概観

イオタ・システムは Developer, Verifier, Debugger, Executer, Prover の五つのサブシステムから構成され、これらは階層的でモジュラーなプログラム作成に必要なすべての情報をもった Data Base によって互いに密接に関連している。(下図参照。二重線はコントロールの流れ、実線はデータの流れをあらわす)



Developer はユーザーとの対話によって、イオタ語でユーザーが作成するモジュールの入力と変更を行ない、内部形式のプログラムをモジュールとモジュール間の関係に関する情報とともに Data Base に保存する。

Verifier は各モジュールの仕様が、そのプログラムの実現によって満たされていることを対話的に検証する。Prover は many-sorted な一階述語論理 (イオタ論理 [9]) のための対話的定理証明システムであり、特に階層的かつモジュラーな構造をもった理論の上の演繹を能率的に行なうように設計されている [4]。(Prover は Verifier から呼ばれる以外に、モジュール間の関係 (type-type 関係の意味論的チェック) を解析するために Developer からもよばれる。また、モジュールの仕様を活用したシンボリックなデバッグにも使える。)

Debugger は、Data Base 内の内部形式プログラムを直接実行するインタプリタと tracer や breaker などの (通常の意味の) デバッグ機能からなり、動的なデバッグを支援する。Executer は、Developer によって生成された内部形式から、オブジェクト・コードを生成、実行するものである。

あとがき

イオタ・システムは現在、京都大学数理解析研究所において、DECシステム20の上で開発がすすめられている。同時にIBM370と日立及び富士通のMシリーズへ移植される。システムの記述にはLISPが適しているが、どのLISPシステムが適当かについては問題があった。結局、イオタ・システムが大規模なものであることから、サポートは少ないが、大きなユーザー・プログラムを効率よく走らせるLISPとしてUtah LISP[7]が採用された。しかし、デバッグやLISP向きのエディタなどの強力な支援システムが必要なので、Utah LISPと同様、Stanford LISP 1.6系統のUCILISP[2]を使って開発が行なわれている。これには複数のTSSジョブを同時に同一端末から走らせるDEC20のOS, TOPS-20のPTYCON[3]が便利に利用できる。(即ち、同一のプログラムを2つのLISPシステムの上で同時に走らせてテストしている。) プログラム開発のための強力な支援システムを持ち(例えば、interlispのように)、開発後は実行に必要な部分のみを独立して効率的に使えるLISPシステムが望ましいが、DEC10, 20の上でも見あたらなかった。将来のLISPシステムの設計にはこの点を考慮に入れていただきたい。

謝辞

イオタ・プロジェクト(特にProver)には香川大学の本田道夫氏も参画している。また京都大学の小島啓二氏の協力もうけている。名古屋大学の金田康正氏には、Utah LISPについて有益な情報をいただいた。

参考文献

1. Atkinson, R., Liskov, B., Scheifler, R.: Aspects of implementing CLU, M.I.T. Computer Structures Group Memo 167, 1978.
2. Bobrow, R. J. et al.: UCI LISP MANUAL.
3. DECSYSTEM - 20 Operator's Guide, 1978.
4. Honda, M., Nakajima, R.: An interactive theorem proving system for structured theories, IJCAI '79.
5. Huet, G. et al.: Environnement de programmation Pascal, Manual d'utilisation. sous siris 7/8. IRIA, 1977.
6. Liskov, B., et al.: Abstraction mechanism in CLU, CACM 8 567-576, 1977.
7. Mavti, J. B. et al.: STANDARD LISP REPORT, Univ. of Utah, 1978.
8. Nakajima, R., Honda, M., & Nakahara, H.: Describing and verifying programs with abstract data types, in Formal Description of Programming Concept (ed. Neuhold) North-Holland Publishing Co. 1977.
9. Nakajima, R., Nakahara, H., Honda, M.: Hierarchical program specification and Verification — a Many-sorted logical approach —, RIMS-Preprint 265, 1978
10. 中島玲二、中原早生、本田道夫: イオタ語とイオタ・システム — 階層的プログ

ラム開発とその検証のための言語と方法とシステム, 第19回プログラミングシンポジウム, 1978.

11. Nakajima, R., Yuasa, T.: Program development with the λ system, RIMS-Preprint, 1979.
12. Parnas, D. L.: The use of precise specification in the development of software, IFIP 77.
13. Yuasa, T.; Module-wise compilation for a language with type-parameterization mechanism, RIMS Preprint-280 1979.
14. Yuasa, T.: Design and implementation of λ language system, RIMS-Preprint 1980.

付録 イオタ語の構文

イオタ語のシンタックスを与える。但し expression と formula については略す。array や record のような built-in type, type などについても略す。現在イオタ語の reference manual が作成中であり、詳細はそれを参照されたい。

<u>part</u> -----> interface -----> -----> specification -----> -----> realization ----->	<u>specification</u> -----> type specification -----> -----> type specification -----> -----> procedure specification ----->
---	--

<u>interface</u> -----> type interface -----> -----> type interface -----> -----> procedure interface ----->	<u>realization</u> -----> type realization -----> -----> procedure realization ----->
--	---

type [synt] interface

```

-----> INTERFACE -----> TYPE [SYNT] -----> module name [synt symbol*] ----->o
    |----->|
    |  

o-----> IS -----> synt symbol* ----->o
    |  

    |<-----, <-----|
o-----> fn-op declaration 1 ----->o
o-----> END INTERFACE ----->
  
```

procedure interface

```

-----> [INTERFACE -----> PROCEDURE -----> module name ----->o
o-----> fn-op declaration 2 ----->o
o-----> END INTERFACE ----->
  
```

type [stype] specification

```
-----> SPECIFICATION -----> TYPE [STYPE] -----> module name [stype symbol*] ----->o
|----->|
|-----> var-declaration 1 ----->o
o-----> AXIOM -----> axiom number -----> : -----> formula ----->o
      ^----->|
      |<----->|
o-----> END SPECIFICATION ----->
```

procedure specification

```
-----> SPECIFICATION -----> PROCEDURE -----> module name ----->o
|----->|
|-----> var-declaration 2 ----->o
o-----> AXIOM -----> axiom number -----> : -----> formula ----->o
      ^----->|
      |<----->|
o-----> END SPECIFICATION ----->
```

type realization

```
-----> REALIZATION -----> TYPE -----> module name ----->o
o-----> REP -----> = -----> type instance ----->o
o-----> fn-body 1 ----->o
      ^----->|-----> op-body 1 ----->|----->|
      |----->|----->|----->|
      |<----->|----->|
o-----> END REALIZATION ----->
```

procedure realization

```
-----> REALIZATION -----> PROCEDURE -----> module name ----->o
o-----> fn-body 2 ----->o
      ^----->|-----> op-body 2 ----->|----->|
      |----->|----->|----->|
      |<----->|----->|
o-----> END REALIZATION ----->
```

fn-op declaration i

```
-----> FN -----> fn-declaration i ----->
|----->|----->|----->|----->|
|----->|----->|----->|----->|
|-----> OP -----> op-declaration i ----->|
|----->|----->|----->|----->|
|----->|----->|----->|----->|
|<----->|----->|----->|----->|
```

```
fn-declaration i |----->|----->|----->|----->|
-----> fn-symbol* -----> : -----> domain i -----> -> -----> range i -----> as definition ----->
```

op-declaration i

```
-----> op-symbol* -----> : -----> op-domain i ----->
```

domain i, range i

```
-----> type instance i ----->
|
|-----> ( -----> type instance i -----> ) ----->|
|               ^
|               |<----- , <-----|
```

op-domain i

```
-----> ( -----> type instance i -----> | -----> type instance i -----> ) ----->
|               ^               ^
|               |<----- , <-----|               |<----- , <-----|
```

as-definition i

```
-----> AS -----> prefix operator -----> type instance i ----->
|
|-----> type instance i -----> infix operator -----> type instance i ----->|
|
|-----> constant operator ----->|
```

var-declaration i

```
-----> VAR -----> variable* -----> : -----> type instance i -----> ; ----->
|               ^               ^
|               |<----- , <-----|               |
|               |<----- ; <-----|
```

type instance 1

```
-----> type instance ----->
|
|-----> & ----->|
```

type instance 2

```
-----> type instance ----->
```

module instance

[type]

```
-----> module symbol* -----> ( -----> type instance -----> ) ----->
|               ^
|               |<----- , <-----|
|               |
|-----> type parameter* ----->|
```

fn-body i

```
-----> FN -----> ^ -----> fn-symbol* -----> ( -----> parameter list i -----> ) ----->o
|
o-----> RETURN -----> ( -----> parameter list i -----> ) ----->o
|
|----->|
o-----> VAR -----> var-list i ----->o
|
o-----> statement ----->o
|
o-----> END FN ----->
```

op-body i

```
-----> OP -----> ^ -----> op-symbol* -----> ( -----> parameter list i -----> | ----->o
|               ^
|               |<----- , <-----|
|               |
o-----> parameter list i -----> ) ----->o
|
|----->|
o-----> VAR -----> var-list i ----->o
|
o-----> statement ----->o
|
o-----> END OP ----->
```

parameter list i, var-list i

```
-----> variable* -----> : -----> type instance i ----->
|
| ^----- , ^-----|
| |----- ; ^-----|
```

statement

```
-----> simple statement ----->
|
| ^----- , ^-----|
```

simple statement

```
-----> assignment statement ----->
|
| -----> if statement ----->|
| -----> while statement ----->|
| -----> for statement ----->|
| -----> operation call ----->|
```

assignment statement

```
-----> variable* -----> := -----> term ----->
|
| --> < -----> variable* -----> > -----> := -----> < ^-----> term -----> > ----->|
| |----- , ^-----| |----- , ^-----| |-----> term ----->|
```

if statement

```
-----> IF -----> unquantified formula -----> THEN ----->o
o-----> statement -----> ELSE -----> statement -----> END IF ----->
while statement |----->|
-----> WHILE -----> unquantified formula -----> DO ----->o
o-----> statement -----> END WHILE ----->
```

for statement

```
-----> FOR -----> nn-var* -----> FROM -----> nn-term ----->o
o-----> TO -----> nn-term -----> DO ----->o
o-----> statement ----->o
o-----> END FOR ----->
```

operation call

```
-----> op-name -----> ( -----> variable* -----> | -----> term -----> ) ----->
| |----- , ^-----| |----- , ^-----|
```

module name

```
-----> module symbol* -----> ( -----> type parameter* -----> : ---> sypte symbol* -----> ) ----->
| |----- , ^-----| |----- , ^-----|
| |----- ; ^-----|
```

fn-name

```
[op] |----->|
|-----> module instance -----> # -----> fn-symbol* ----->
| |----- [op] ----->|
|-----> ^ -----> fn-symbol* ----->|
|-----> [op] ----->|
```



本 PDF ファイルは 1980 年発行の「第 21 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、権利者の検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、一定期間の権利者検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>