

超時空プログラミングシステム Uranus

電子技術総合研究所

中島 秀之

Hideyuki Nakashima

1. はじめに

超時空プログラミングシステム Uranus は、知識表現を中心とした総合プログラミングシステムである。特に超時空の名は、多重世界機能を用い、様々の概念の空間や、時間を取り扱えるところに由来する。

Uranus (Uranus) の名はギリシャ神話に登場する世界の支配者である神の名(天王星の英語名としても有名である)から取った。また、システムの名前としては、Universal Representation-Aimed Novel Uranus System というリカーシブな定義を特徴とする。

Uranus は Prolog を基本とし、それに Lisp の持つ制御構造や、多重世界機能を追加し、拡張することにより、知識表現に適したプログラミング言語となっている (Prolog/KR [中島1982] の発展形である)。Prolog を完全に包含した言語であるが、同時に関数的な表記も可能で、Lisp とほとんど変わらぬ表現もできる。ただし、両者は全く同一の言語基盤(ロジック)の上に構築されており、従来のシステムにありがちな、複数言語の混合体ではない。従って、宣言的記述から手続き的記述までをプログラムの目的に合せて使い分けることが可能である。

ある概念を表現するには、その概念に関する表明の集合を用いる。そして、この集合をひとつの世界にまとめることにする。このような世界の間の関係を記述することにより、概念間の関係を表わすことができる。概念間の関係というものは、概念にとってメタレベルに存在する。このメタレベルの記述が同一の言語の枠内で行なえるのが Uranus の特徴である。メタレベルの記述を、同一の言語で行なうことは、その更にメタレベルの記述も同一の枠組みでとらえられるということである。このようにして Uranus は自然言語の持つ、自己に関する記述(例えば、「この文は間違っている」など)と同等の力を持つことになる。

本発表では、Uranus の多重世界機構と、それを用いた表現について概観する。

♫ Uranus は“ウラヌス”あるいは“ウラノス”と発音する。決して“ウラナス”(裏茄子)と読んではいけない。

2. 多重世界機構

Uranus のプログラムはいくつかの互いに独立な世界より構成される。ここでいう世界とは述語の集合のことである。述語の定義は世界ごとに異なっていてよく、また全部の世界で定義されている必要もない。各々の世界で定義された述語は外側からは見えない。それらを起動するにはwithというプリミティブを用いて、その世界に入る必要がある。

多重世界機構を用いることにより以下のようなことが可能になる。

(1) ひとつの概念に関する表明 (assertion)を一箇所に集めることができる。この場合、世界の名前が記述される概念の名前に対応する。[中島1984, Nakashima 1984]

(2) 状態の遷移を時系列としてとらえる場合に、各々の世界を、ある特定の時刻における世界の断面と考えることができる。

ここでは、これらの多重世界機構の果たすべき役割と、世界の間係を操作するためのプリミティブに関して考察する。

2.1. with

ある世界に入るにはwithというプリミティブを用いて、

(with 世界名 . 述語呼出し)

のようにする。述語呼出しの部分には普通の述語の呼出しの他、定義など任意のものが任意個書ける。世界の名前は任意のものを使ってよく、その世界が既に存在すれば、それが使われ、存在しない場合には新たに作られる。

例えば、Japaneseの世界でfirst-numberという述語を実行するには

(with Japanese (first-number *x))

とする。おそらく*x= いち となるであろう。同様に、

(with English (first-number *x))

を実行すると*x= one となり、

(with French (first-number *x))

なら*x= unという具合である。このように同じ述語でも、世界ごとに定義が変わりうる点に注意されたい。

述語呼出しの部分には普通の述語の呼出しの他、定義など任意のものが任意個書ける。従って、先程のfirst-numberを定義するにもwithを用いることになる。

(with Japanese (assert (first-number いち))

(assert (second-number に))

...)

(with English (assert (first-number one))

(assert (second-number two))

...)

世界は何重にもネストして用いることができ、その場合内側の世界からは外側の世界の

定義が（原則として）全部見える。この世界間のネスティングは定義時ではなく実行時に決まる。すなわち、ある述語が実行されると、世界のネストを内側から順に走査してその定義を探す。例えば世界Aではpという述語が、世界Bではqという述語が定義されているとすると

```
(with A (with B (and (p *x) (q *x))))
```

においてはp、qはそれぞれ世界A、Bの定義を参照することになる。バックトラックの際にはさらに外側の定義を調べる。したがって、同じ述語がいくつかの世界で定義されている場合には、内側のものから順に試みられることになる。例えば述語pが、世界A、B、Cでそれぞれ

```
(with A (assert (p a)))
```

```
(with B (assert (p b)))
```

```
(with C (assert (p c1)))
```

```
(assert (p c2)))
```

のように定義されているとしよう。これを

```
(with A (with B (with C (p *x))))
```

のように呼び出すと

```
*x=c1, c2, b, a
```

の順で解が得られるし、

```
(with C (with B (with A (P *x))))
```

のように呼び出すと、解は

```
*x=a, b, c1, c2
```

の順になる。

外側の世界の定義を隠してしまうにはassertではなくdefineを用いる。assertが、これまでの定義に追加する動きを持っているのに対し、defineはこれまでの定義を消去してしまう動きがある（ただし、外側の世界を変えることはしない）。先程の例で世界Bにおけるpをdefineを用いて

```
(with A (assert (p a)))
```

```
(with B (define p ((b))))
```

```
(with C (assert (p c1)))
```

```
(assert (p c2)))
```

のようしておくと、

```
(with C (with B (with A (P *x))))
```

の呼び出しの解は

```
*x=a, b
```

となり、Cの定義が見えなくなる。別の節を探すときにdefineに遭遇すると外側の世界の探索を打ち切るのである。

2.2. 概念の記述

ある概念に関する表明 (assertion)、すなわち概念の満たす関係の集合によってその概念を規定することができる。例えばペンギンという概念は

```
(assert (HAS wing))  
(assert (CARDINARITY wing 2))  
(assert (HAS eye))  
...  
(assert (INHABITANCY Antarctica))  
(assert (COLOR black))  
(assert (COLOR white))  
...
```

のようなペンギンに関する表明の集合により定義できる。そこで、これらの表明を一つの世界としてまとめ、この世界がペンギンという概念に対応すると考えることができる。

ただし、ここで注意しておきたいのは、これはプログラマの側の思い込みであって、システム自身はそのような認識は持っていないという点である。システムにとって世界というのはwithというプリミティブによって開くことのできる述語の定義の集合に過ぎない。従って次節(2.3.)のように仮想世界を表わすのにも全く同様の機構が用いられる。

2.3. 仮想世界としての多重世界

積木の世界など、変化する世界を対象として取り扱う場合には、その世界に関する表明自身を変化させてしまうのが便利である。例えば積木を別の積木の上に動かす(図1)述語moveは

```
(assert (move *x *from *to))  
  (retract (on *x *from))  
  (assert (clear *from))  
  (retract (clear *to))  
  (assert (on *x *to)))
```

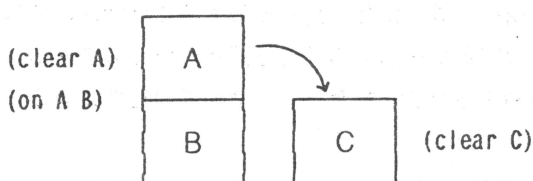


図1 積木の移動

のように、古い状態に関する表明を retract し、新しい表明を assert するものとして定義できる。

しかしながらこの解法ではバックトラックの際に問題が残る。assertやretract はバックトラックの際に元に戻らないからである。例えば、

```
(move A B C)
```

において、Cの上にすでに何か別の積木が乗っていたとすると(clear C) がないので

```
(retract (clear C))
```

が失敗する。しかしながら、この場合にもそれ以前の二つのassertとretract は元には戻

らない。すべてが旨くいくことをテストしてから実行に入れば、このように途中で失敗することは防げるが、一般にはテストと実行はほとんど同じ動作の繰り返りで二度手間になるので、できれば一回ですませたい。

このような場合に、多重世界機構が状態の変化を表わすのに使える。様相論理の場合のように、公理の変化は世界の変化と考えてよい。積木の移動と同時に別の世界に移ったと考えるのである。assertやretract は、この新しい世界で行なわれ、元の世界は不変のまま残る。積木AをBからCに動かして到達する世界をmoveA-B2C と名付けることにすると

```
(with moveA-B2C
  (retract (on A B))
  (assert (clear B))
  (retract (clear C))
  (assert (on A C)))
```

が成立する。変化は新しい世界のみにかかるので、動作が成功した場合にのみ新しい世界に移ることにしておけば、バックトラックして古い世界に戻ったときには、そこでは元のままの状態が保存されている。いく通りかの動作が可能な選択点では、したがって、

```
(or (with branch1 <action 1> <further actions>)
    (with branch2 <action 2> <further actions>)
    (with branch3 <action 3> <further actions>)
    ... )
```

のようにしておけばよい(図2)。branch1 で <action 1> を試みて失敗してbranch2 に来ても、こちらの世界では<action 1>の効果は残っていない。

一般に

```
(with nil <action 1>
  (with nil <action 2>
    (with nil <action 3>
      ...
    )))
```

で動作の列を表わすことができる。ここでnil というのは名前のない特別の世界で、毎回新しいものが自動的に作られる。

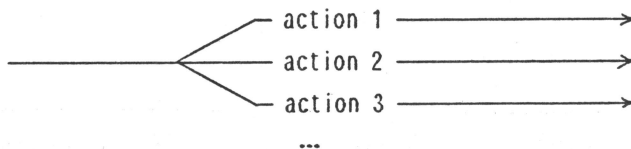


図2 世界の分岐

3. メタレベルの記述

Uranusでは、世界間の関係を記述することができる。この世界というのはプログラムの集合のことであるから、Uranusのプログラムにとってはメタレベルの概念である。このメタレベルの概念を、以下に示すように、再びUranusのプログラムから操作できるということは、何層にもわたるメタ*レベルを単に1層の記述言語に圧縮できることを意味する。⁴

3.1. 概念の階層構造の記述

まずは、概念の階層構造と、それらの間の属性の継承の表現を例としてこの機能を見てみたい。

Uranusでは、属性（プログラムを含む）の継承のルートは実行時の世界のネスティングの状態によって決定される。そこで例えば、PENGUIN の環境を設定するには

```
(with OBJECT
  (with ANIMAL
    ...
    (with BIRD
      (with PENGUIN
        (NUMBER-OF-WINGS *n)))...))
```

のようにwithを何重にも重ねる必要が生じる。これは煩わしい（心理的問題であって、言語の記述力の問題ではないが…）ので、世界間の関係を表わすメタ知識として実行時に望む環境に展開してくれる述語を定義したい。これには様々な方法があるが、一例を示すと、

```
(assert (AKO *sub *super)
  (assert (WITH-WORLD *sub *p)
    (WITH-WORLD *super (with *sub *p))))
(assert (WITH-WORLD *x *p) *p)
```

のように定義すればよい。内側のassertはAKO が実行されたときにダイナミックに WITH-WORLD を定義するためのものである。また、2番目のWITH-WORLDの定義は階層の一番上ではトップレベルの環境を用いることを指示している。こうしておくと、

```
(AKO PENGUIN BIRD)
(AKO BIRD ...)
...
(AKO ANIMAL OBJECT)
```

⁴一般にメタレベルというのはもう少し広い意味に使われるが、ここでは世界間の関係の記述という点に限る。

の実行により必要なWITH-WORLDが定義され、

```
(WITH-WORLD PENGUIN (NUMBER-OF-WINGS *n)))
```

の実行時に、先に出てきたようなwithのネスティングに展開される。

このAKO とWITH-WORLDは世界の間の関係を記述するメタ知識と考えてよい。

実行時に毎回ネスティングを作り出すのは効率の面でよくない。そもそも概念の階層構造は静的なものであるから、完全に動的なネスティングにマップする必要はない。そこで、**Urans**ではwithinという述語も用意してある。withinは第1引数として世界のネスティングを表わすリストを取る。リストの先頭が一番内側の世界で、それから順に外側へとネストする。第2引数以降はwith同様、任意の述語呼び出しが書ける。例えば

```
(within (A B C) (p *x))
```

は

```
(with C (with B (with A (p *x))))
```

とほぼ等価である。さて、このwithinを用いると

```
(with PENGUIN (assert (supers PENGUIN BIRD ... ANIMAL)))
```

としておけば、

```
(with PENGUIN (within [supers] (NUMBER-OF-WINGS *n)))
```

として一度に実行できる。ここで[supers]は実行可能パターンで、(supers *s) を実行した*sの部分がここに書いてあるのと等価である。一般にWITH-WORLDは

```
(assert (WITH-WORLD *w *p)
```

```
(with *world (within [super] *p))))
```

となる。これは第二のメタ知識の記述法である。このように世界間の関係を記述する方法は幾通りもありうる。

3.2. 仮想世界間の関係の記述

仮想世界の考え方はCONNIVER [Sussman et al. 1972]のコンテキスト機構に酷似している。様々な仮想世界を同時に扱い、それらの間の比較をしたりすることが可能である。問題解決という側面からみると、バックトラックを伴う通常の実行と仮想世界の機能はレベルが異なるだけで、実は同じものであると見なすこともできる。以下にその対比を示す。

通常実行	仮想世界
節（表明） 変数の値 バックトラック	世界 表明 世界からの脱出

通常実行では、ある選択枝が選ばれたときにユニフィケーションによりその枝に相当する

ように変数の値が定まる。これに対し多重世界の考え方では、ある世界が選ばれたときにその世界に相当する表明の集合が有効になる。共に、ある問題に対応した、変数の値あるいは表明の集合（状態を示す）が解となる。

多重世界の利点はバックトラックなしにその世界から脱出できることである。バックトラックするとすべての結果が消えてしまうが、多重世界の場合は一旦外に出ても状態は保存される。そのため選択枝間の比較が可能である。この選択枝間の比較を行なうためには、選択枝（仮想世界）間の関係を記述するメタ知識が必要である。この記述を適切に行なうことができれば、バックトラックを必要としない問題解決システム [de Kleer 1984] の実現も可能である。ただし、この際には世界の集合のようなものを陽に表現する機能が必要である。例えば

```
(with A (assert (fallible *x) (god *x)))  
(with B (assert (fallible *x) (human *x))  
  (assert (god Uranus)))
```

という二つの世界があった場合に、どちらか片方の世界だけで

```
(fallible Uranus)
```

を証明できないが、両方の世界を合わせると証明可能である。この場合、この結論はどの世界に属するものであろうか。強いて記述するなら

```
(within (A B) (assert (fallible Uranus)))
```

とでもなろうが、現在のwithinの仕様はこの気分に対応していない。

4. 問題点

ここでは将来の課題としてUranusが抱えている問題点に関して述べる。

4.1. 問題1

多重世界機構を仮想世界として用いる場合には、assertやretract がどの世界に対してなされるべきかが問題になる。仮想世界のネストを作っている理由は、元の世界を変更しないためであるから、変化は一番内側の（つまり、一番新しい）世界でおこななければならない。assertの場合は外側の世界との和になるので、一番内側の世界にassertしておけばよい（ことが多い）。ところが、retractの場合は、一番内側の世界にはretractすべきものもともと存在しない。

Uranusでは、assert、define、definition、およびretractの、述語の定義を操作するメタ述語は一番内側の世界に対して働くものとしている。従って、一番内側の世界に定義がない場合には、外側にそれがあっても見に行かず、retractが失敗する。definitionを取ってくるときやretractする場合に、それらが効果を持つためには、その述語が定義されている世界でそれらを実行する必要がある。ところが、この問題に関しては先にも述べたように、一番内側の世界でretractしてくれなくては意味がない。そこで、Uranus

ではretract1という述語を導入した。これは外側の定義をすべて一番内側の世界にコピーした後にretract を起動する。また、バックトラックの際に外側の世界を見に行かないようにdefineの場合と同様のマーカーを最後に付加しておく。retract1により上記の問題は一応解決されるが、一番内側の世界だけが単独で用いられた場合にはセマンティクスがおかしくなる。retract1は、あくまで仮想世界として

(with nil ...)

の中でのみ使うべきである。

この問題の根本的解決策としては負のassertion を考える必要がある。何もない世界からretract すると、それは負のassertion となり、外側の世界のassertion を打ち消すのである。様相論理的視点から言えば、こちらの方が“論理”的であろう。

4.2. 問題2

また、更に図3のような状況も考えられる。これらは共に、世界がいくつか集まった構造自身がひとつの仮想世界を構成している例である。図3 aは新しい仮想世界において、世界間の構成が変化した場合、図3 bは新しい仮想世界において一部の世界が変化した場合である。aの場合は世界間の構成を記述する述語（例えば3.1.で述べた WITH-CLASS や AKO のみを一つの仮想世界としてまとめておけば（図4）、その世界の変化として記述できるので、bの問題のサブセットと考えることもできる。

bの場合の解決策としては、仮想世界1のなかの世界Aをコピーした世界A'を仮想世界2のなかに作ることである。しかし、世界のコピーは高くつくので、2.3.の例のようにその差だけを記述する方法を取りたい。

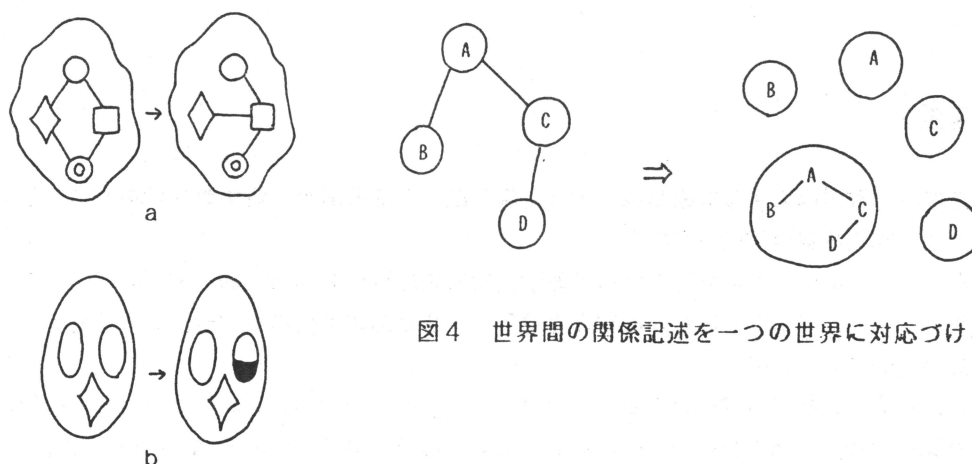


図4 世界間の関係記述を一つの世界に対応づける

図3 世界を要素とする仮想世界の変化

4.3. 解決に向けて

問題1と問題2はまとめて解決すべきものである。すなわち、世界と世界の関係だけでなく、それらの間の差（正・負ともに）を静的に記述するプリミティブが必要である。例えば、世界W1は世界W2とほとんど同じだが、(on A B)が成立しない点だけが異なる、というような記述が（新しい仮想世界の中だけで）可能になれば、いくつかの世界を含む仮想世界の変化を記述することが可能になる。assertやretract を、実際に世界の中身を変えてしまう操作でなく、世界自身を変えるための様相オペレータと考えるアプローチが有効かもしれない。Warrenは、これに関して assert の側のみについて提案している [Warren 1984] が、これをretract の側にも拡張したものが望まれる。

5. おわりに

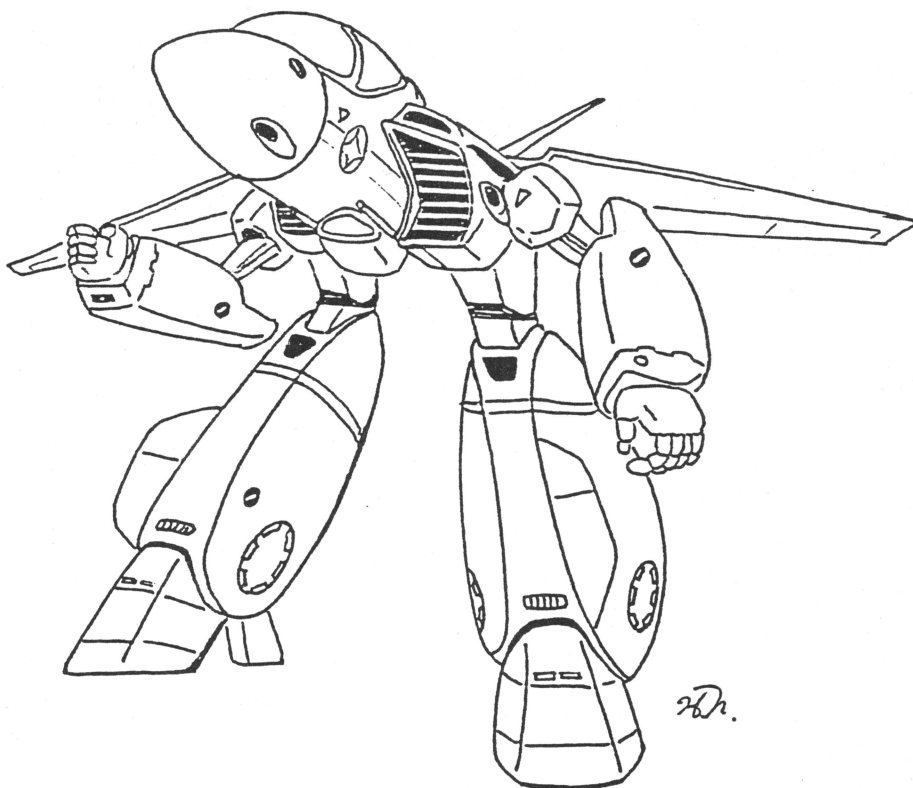
Uranusの多重世界機構と、それを用いた、概念の階層構造や仮想世界の記述に関して述べた。世界間の動的なネストにより、実行時の世界の構造を指定することができる。これはUranusのプログラムとして記述できるので、自分自身で自分の環境の記述が（不十分ながら）できたことになる。ただし、今のところ世界の間相違を静的に記述する機能に欠けている。これは今後の研究課題である。

ロジック・プログラミングに概念の階層構造に相当する機構を導入する試みとしては、ESP [Chikayama 1984] やMandala [Furukawa et al. 1984] が存在する。ESP は、効率は良いが、静的な関係しか扱うことはできない。一方Mandala はdemo述語 [Bowen 1982 < 国藤他1983] を基本としており、ここで述べたような、メタレベルの記述能力は高いと考えられる。ただし、あまりにも柔軟すぎて、効率のよい実行は危ぶまれる。Uranusのアプローチは両者の中間と言えよう。

参考文献

- 国藤進他：“Prologによる対象知識とメタ知識の融合とその応用” 情報処理学会知識工学と人工知能研究会30-1 (1983)
- 中島秀之：“Prolog/KR の概要” 情報処理学会記号処理研究会18-5, pp. 69-74 (1982)
- 中島秀之：“Prolog/KR における知識表現” 情報処理学会第28回全国大会論文集, pp.1139-1140 (1984)
- K. A. Bowen, R. A. Kowalski: “Amalgamating Languages and Metalanguages in Logic Programming” in Logic Programming, eds. Clark and Tarnlund, Academic Press, pp. 153-172 (1982)
- Takashi Chikayama: “Unique Features of ESP” Proc. of the International Conference on FGCS, pp.613-622 (1984)

- Johan de Kleer: "Choices without Backtracking" Proc. AAAI-84, pp.79-85 (1984)
- Koichi Furukawa et al.: "Mandala: A Logic Based Knowledge Programming System"
Proc. of the International Conference on FGCS, pp.613-622 (1984)
- Hideyuki Nakashima: "Knowledge Representation in Prolog/KR " Proc. of The 1984
International Symposium on Logic Programming, pp.126-130 (1984)
- Gerald J. Sussman, D. McDermott: "From PLANNER to CONNIVER, a Genetic Approach"
Proc. FJCC 41, AFIP (1972)
- David Scott Warren: "Database Updates in Pure Prolog" Proc. of the Inter-
national Conference on FGCS, pp.244-253 (1984)



本 PDF ファイルは 1985 年発行の「第 26 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、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。

https://www.ipsj.or.jp/topics/Past_reports.html

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、一定期間の権利者搜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>