ソフトウェア工学 85－2
（1992. 5. 21）

母国語プログラミングへの方式，実践とその効果

中川正樹，玉木裕二，早川栄一，曽谷俊男
東京農工大学工学部

本稿は，母国語（我々の場合は日本語）によるプログラミングの理念，母国語の使用が制限されないプログラミング環境の実現方式，母国語プログラミングの実践，そして，その効果について述べる．なお，本稿で言う母国語プログラミングとは，曖昧性を有する自然言語で処理（論理）を記述するのではなく，文字種として母国語が制限なく使用できる既存プログラミング言語によるプログラミングを意味する．母国語プログラミング環境の実現では，システム記述言語処理系の母国語化が前提となった．我々は，上記処理系ソースプログラムの最小修正で系統的に母国語化を行った．この環境で相当規模のソフトウェアを開発し，母国語プログラミング環境のソフトウェア開発への影響を調査した．母国語プログラミング環境は，概念設計からプログラムに至る段階的詳細化を可能にした．仕様書におけるキーワードは最終的プログラムに識別子などの形で反映されている．また，仕様書とプログラムの一体化は保守性にも寄与している．生産物としての日本語プログラムでは，処理の日本語による言換えのためのコメントは減り，モジュールの仕様を記述するコメントが大部分を占めるようになった．別に，プログラムの可読性評価実験を行い，日本人にとっては日本語プログラムの可読性が高いことを検証した．本稿ではさらに，母国語プログラミングのその他の利点についても論じる．

Philosophy, Practice and Effect of Programming in a Mother Tongue

Masaki Nakagawa, Yuji Tamaki, Eiichi Hayakawa, Toshio Souya
Dept. of Computer Science, Tokyo Univ. of Agriculture and Technology

2-24-16 Naka-cho, Koganei, Tokyo, 184, Japan

This paper presents philosophy of programming in a mother tongue particularly in Japanese, strategy to construct a computer system where Japanese can be used for any purpose, practice of programming in Japanese and discussion on its effect. The key of the system Japanization was that of a language compiler for systems implementation, which was made systematically with the least modification of the source program.
A considerable size of software was developed on this system. Smooth stepwise refinement from specifications to programs has been realized. Keywords in specifications remain as function, data names and so on in programs. Coherence between specifications and programs eases maintenace. A notable difference was observed in software products. Japanese programming dispenses with paraphrase of program operations into Japanese comments and encourages to provide module specification. Experiments were made to show that programs are more readable in Japanese than in English for Japanese people. Several advantages of programming in a mother tongue are also discussed.

1．はじめに

プログラミングは知的な活動であり，人間の思考は母国語に依存している．また，プログラミングの対象が，地域的，民族的，社会的，あるいは，文化的問題に広がるにつれ，対象や処理を名付けるのに日本語が使えないと不便である．さらに，ソフトウェアは書籍と同様に知的財産を形成しつつある．プログラムを読む割合がますます大きくなっている．ソフトウェア工学でも，プログラムを新しく作るより，プログラムを読むことに多くの労力が費やされている．当然，日本語の方が読み易い．

ソフトウェアの生産性を向上させるために，ツール，方法論，パラダイム，環境などが提案され，議論され，そして，実践されてきた．しかし，非英語圏の人間にとって，母国語の使用が制限される環境では，それらが効果を発揮する以前に大きな問題がある．このような理由から，識別子，メッセージ，コメントなどを含めて，母国語で制限なくプログラミングできる計算機システムを開発した．ここでいう“母国語プログラミング”とは，曖昧性を有する自然言語で記述するのではなく，プログラミング言語の文法の範囲内で，母国語の使用を自由にすることを意味する．

プログラミング言語だけでなく，計算機システム全体をオペレーティングシステムのレベルから日本語化した．母国語化を行なう場合，各アプリケーションや一言語のレベルだけでとどめる訳にはいかない．これは，他アプリケーションとの組合せや他言語プログラムとのリンケージの必要性を考えれば明かである．例えば，ある言語で日本語識別子が使えても，リンカやソフトウェアツールが対応していなかったら，日本語識別子を使うことは逆に障害になる．

日本語を扱う計算機システムの中核をなすのは，日本語で記述したシステムプログラムのコンパイラと，日本語を文字コードとするオペレーティングシステムである．文字コードは，次章で述べる理由から，制御コードを含めた２バイト固定長コード系を設定した．システム記述言語としては，言語Ｃを選択した[1]．我々は，言語Ｃコンパイラを独自で開発し[2, 3, 4]，これを CAT (C compiler developed at Tokyo Univ. of Agri. and Tech.) と名付けた．また，日本語オペレーティングシステムを作成し，OS/omicron と名付けた[5, 6]．母国語化の一環で，日本語入力[7, 8]や日本語文書出力システム[9,10]を研究開発した．仮想マシン環境の開発[11]と市販ワークステーションへの移行を経て，現在は，システムソフトウェアの研究とともに，アプリケーションの研究にも実用し，日本語プログラミングを実践している．本稿で，日本語プログラミングの実践例として取り上げる手書き日本語文字認識の研究も，この計算機システム上で行っている．

今日，ほとんどのアプリケーションは，日本語のインタフェースを持つことが一般的になってきた．商用システムでは，既存システムのコード系に日本語文字コードを混在させて日本語を表現している．しかし，それらの日本語化は多くの場合，不完全で，部分的である．このような実現方法では，非本質的な処理に煩わされたり，日本語の使用が制限されることになる[12]．一方，既存システムの拡張ではなく，日本語を表現できるコード系を設定して日本語化した報告として，日本語Pascal[13]，日本語CLU [14]がある．しかし，日本語化の効果については報じていない．我々は，オペレーティングシステムレベルからの日本語化を最初に報告している[5]．本稿では，その日本語化方式を我々のシステムに限定しない形で提示し，さらに，日本語プログラミングの効果を定量的に示すものである．

90年代に入り，米国製オペレーティングシステムのマルチバイト化が話題になっている．背景には，米国製ソフトウェアの海外での売上の急増がある．もちろん，各国に合わせる国際化や地域化をしたうえでのことである[15]．市場の成長率から，米国メーカは，米国内に遅れずにソフトウェアを海外でも発売したい戦略がある．ところが，１バイトコード系で開発したソフトウェアを混在コード系に直すにはソフトウェア技術者の労力を必要とし，日本などの市場に投入するのに相当な遅れを生じてきた．そこで，海外仕様としては Unicode[16]で開発し，それを翻訳の専門家が各国の言語や事情に合わせればよいと考えるのは自然である．あるいは,ASCII環境で開発し，それを Unicodeにコード変換後，各国対応にしてもよい．Unicode への変換は，後で我々の場合で示すのと同様に，元のプログラムで文字型が１バイトであることに依存したコーディングをしていない限り，文字コード変換だけの問題である．理論的には，変換先は統一コードである必要性さえなく，固定長コードでありさえすればよい．

我々は，エンドユーザのためのインタフェースだけでなく，ソフトウェア開発のために母国語化が必要であると主張してきた．そのために母国語が制限なく使える環境が必要となり，米国メーカに先んじオペレーティングシステムレベルからのマルチバイト化を実現した．

多字種文字を扱うのに，マルチバイト固定長コードを採用すれば，本稿で示すように計算機システムの母国語化は系統的に実現できる．母国語が使えれば，ソフトウェア開発の生産性や品質の向上が期待できる．さらに，固定長コード体系間ならば，ソフトウェアを翻訳するのはプログラミングの問題ではなくなり，ソフトウェア技術者の労力を必要としない．以上の帰結として，各民族が優れたソフトウェアを自分達のために円滑に相互利用できるようになるはずである．

２バイト固定長コードの採用に対し，これまで何度かその経済性への疑問がなげかけられてきた．本論文では，この問題に対し，いくつかの視点から答える．

2．母国語によるプログラミング環境

2.1 完全な日本語化に向けて

制限のない日本語プログラミング環境を提供するためには，システムのコード体系から見直す必要性を認識した．我々は，完全な日本語化の最善の方法として，システムソフトウェアの内部コードに，2バイト固定長の日本語文字コードを採用することを，CAT 初版の開発当時から計画した[17]．

大きな文字セットを持つ言語のために，2バイトコード系は ISO で定義されており，日本語文字コードはその一環で，JIS X 0208として定義されている．日本のディスプレイやプリンタは，この文字コードを基本としている．このコード系には，仮名，漢字だけでなく，ASCII 文字や記号，ギリシア文字，さらにロシア文字さえも含まれている．このため，文字を表現するために，非常に大きな自由度が得られる．

システムの文字コードを2バイト系に変えてしまうと，1バイトコード系のソフトウェアは変更なしには使えない．しかし，2バイト固定長コード系への変換は，元のプログラムが文字コード長に依存したプログラミングをしていない限り，ソースプログラムのコード変換と再コンパイルだけで済む．一方，1バイト2バイト混在コード系への対応では，文字列処理の部分をすべて書き直さなければならない．

文字コードを2バイト固定長コードで表現することによる完全な日本語化は，システム開発の見地からも効率的であった．実際，ASCII 環境で CATの初版を開発するときには相当の労力を費やしたが，その初版CAT を完全に日本語化するには，初版CAT の開発者以外の学生5名で，2週間しか要しなかった[4]．

2.2 日本語2バイト固定長コード系

OS/omicron では，文字コードとして，JIS X 0208 の日本語2バイトコードを採用した．一方，JIS X 0202 は，ASCII コードに相当する JIS X 0201 の1バイトコード系に，JIS X 0208 の2バイトコードを混在させる構造を定義している．これら3つの規格は，1バイトコードに2バイトコードを混在させることを仮定している．というのは，JIS X 0208では2バイトの制御コードはなく，JIS X 0201の制御コードを使わなければならないからである．

そこで，完全な2バイト固定長コード系にするために，1バイトの制御コードの前に，NULL$(00)_{16}$ を挿入した．JIS X 0201 によれば，NULL はどこにでも挿入してよいことになっている．これにより，このようにコード系を修正しても，依然規格に適合していることになる．この修正したコード系を，2バイト固定長 JISコード系と呼び，断わらなければ，単に JIS と略記する．

2.3 JIS コード系のシステムソフトウェアの開発

2.3.1 完全日本語化への方針

2バイト固定長 JIS コードの計算機システムを作成するために，あるコード体系のシステムから別のコード体系のシステムを生成する，一般的で系統的な手法を考察した． ASCII から JISへの変換を例にして述べる．まず，システム記述言語処理系の文字型を，1バイトから2バイトに変換し，ASCII 文字セットから日本語文字セットに拡張する．そして，システムソフトウェアのすべてのソースファイルの文字コードを，ASCIIコードから2バイト固定長JISコードにコード変換する．必要に応じて，メッセージやコメント，識別子などを日本語に翻訳する．処理やデータ構造の変更の必要はない．最後に，日本語化したシステム記述言語処理系で再コンパイルする．英語によるプログラミング環境が，英語によるプログラミングと英語を文字コードとするシステム記述言語処理系によって実現されるように，日本語によるプログラミング環境が生成される．混在コード系よりもむしろ，マルチバイト固定長コード系の方が，簡単にしかも系統的に母国語化できる．Earleyらの定式化[18]に文字コードと文字セットの記述を追加して，日本語化の方式を図1に示す．

ただし，日本語化を系統的に行なうためにはシステムソフトウェアのコーディングにおいて，文字と1バイトデータを明確に区別しておかなければならない[12]．

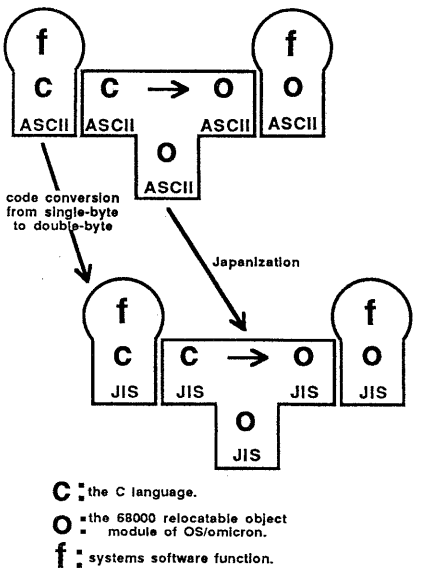

図1　システムの日本語化戦略

2.3.2 システム記述言語コンパイラの完全な日本語化

言語Cコンパイラ CAT はもともと，ASCII 環境で開発した．これを ASCII CAT とする．すでに述べたように，この言語処理系を日本語化すれば，システム全体を日本語化できる．そこで，CAT の日本語化も系統的に行ないたい．つまり，CAT を日本語化するとき，ソースプログラムの修正は最小限に抑えたい．CAT の日本語化は，次のステップによった．

(1) JIS コード系による ASCII 文字セットを受け付ける meta1 CAT の生成

(2) JIS コード系による JIS 文字セットを受け付ける meta2 CAT の生成

(3) 文字コード，文字セットが共に JIS である JIS CAT の生成

上記の過程を，図2に図示する．

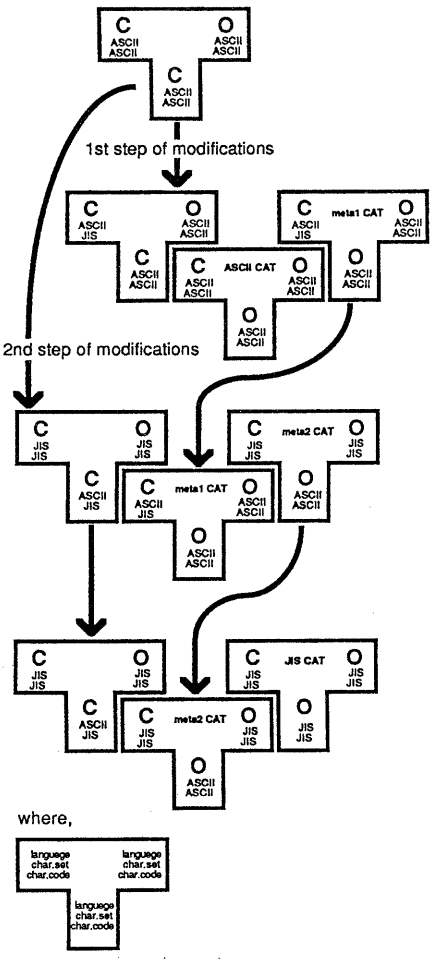

図2　JIS CAT を生成するための
CAT の複数の版のインタラクション

2.3.3 更なるマルチバイト化

多字種言語への対応でマルチバイト固定長コード体系を採用しておけば，それに字種追加や属性追加が必要になって一層長いバイト長になっても，以上の手続きを繰り返せばよい．

2.4 2バイト固定長コードの経済性

英数字や制御コードまで2バイトで表現すると，記憶装置と転送のコストを最悪で2倍にするという意見がある．しかし，混在コードでは文字列スキャンが複雑になり，CPU への負荷が増える．

問題は人的コストである．混在コードを処理するためにプログラミングと保守のコストはますます大きくなりつつある．メモリや転送の相対的コストはますます小さくなる一方で，人的コストは増大の一途をたどるばかりである．

システムソフトウェアのマルチバイト化は一見，過大なコストを要するように見えるが，ソースプログラムさえあり，文字コード長に依存したプログラミングさえしていなければ，ここに示したようにきわめて系統的に行なえる．反対に，混在コードはその場しのぎになっても，それ以後開発されるすべてのソフトウェアで混在コードに対処する負荷を強いることになる．

3．日本語プログラミングの実践

3.1 開発したソフトウェアの概要

以上述べてきた日本語プログラミング環境で，最初に開発した相当規模のプログラムは，オンライン手書き日本語文字認識システムである．種々のノイズや手書き変形に対処するために，また，2000以上の多様な文字パタンを認識するために，プログラムは大きくなり言語Cで 30,000 行を超えている．ある程度の大きさのプログラムを取り上げるのはマクロな性質を議論するためである．

本システムの日本語プログラム例を付録に示す．

3.2 概念仕様から日本語プログラムへ

本システムの研究開発では，1ページ約1600字の用紙で40ページの概念仕様書から，同用紙で約 90 ページの設計仕様書，同 300 ページの関数仕様書，および部分的ではあるが約 20ページの PAD による図示表現をへて，日本語プログラムへと詳細化した．OS/omicronの日本語プログラミング環境は，概念設計からプログラムに至る段階的詳細化を可能にした．仕様書におけるキーワードは最終的プログラムに関数名，変数名，データ構造名などの形で反映されている．また，仕様書とプログラムの一体化は保守性にも寄与している．仕様書作成やプログラムの初期コーディングでは，ワードプロセッサも利用した．

3.3 コメントの役割

日本語プログラミング環境で開発したプログラム（日本語版）と，日本語がコメントにしか使えない過渡的環境で以前作成した約 15,000 行のプログラム（不完全日本語版）の比較を行った．この結果，コメントがどこでどのように付けられているかに顕著な差異が現れた．表１に，両者のコメントの統計を示す．不完全日本語版では，ソーズプログラムの右側に処理の意味を日本語で説明する（言い換える）ためにコメントが付けられていた．一方，日本語版では，90% 以上のコメントが，関数やモジュールの先頭で仕様を記述するために使われていた．

日本語で識別子に名前付けできれば，処理内容をコメントとして日本語で言い直す必要がなくなり，関数やモジュールの仕様を書くことにつながった．

4．日本語プログラムに対する評価実験

4.1 可読性の評価実験

プログラムの可読性に関して実験を行った．まず，コメントを全然付けない２種類のプログラムを用意した．一つはコメントに関する統計を集めるプログラム（約150行）であり，もう一つは指定された前後関係で出現する数字列を検出するプログラム（約110行）である．さらに，それぞれについて，日本語で記述したものと，英語で記述したものを用意した．英語版については，英語を母国語とする修士学生に手を入れてもらった．これらを，コ＿日，コ＿英，数＿日，数＿英，とする．

学部４年生，大学院生を被験者とし，これらを G1 と G2 の２つのグループに分けた．さらに G1 を G1.1 と G1.2 に，G2 を G2.1 と G2.2 に分けた．グループ分けでは，過去のプログラム経験や研究分野を参考にして，能力や知識でグループ間に偏りを生じないよう配慮した．G1.1 の被験者には コ＿英 と 数＿日 をこの順に，G1.2 には同じものを逆順に，G2.1 の被験者には コ＿日，数＿英 をこの順に，G2.2 には同じものを逆順に与えた．被験者には，それぞれのプログラムについて，読むのに要した時間，理解を確認するための設問に答えるのに要した時間を記録するように指示した．表２に，設問に正しく答えた被験者について結果を示す．グループ間で被験者数に不均衡があるのは，正解が得られなかった被験者を除いたためである．プログラムを読む時間と，設問に答える時間を別個に見た場合，有意なことは言えない．これは，プログラムを十分理解して設問に答えるタイプと設問を見てからプログラムを見直すタイプの個人差が大きいためであろう．そこで両方の時間の合計で見ることにする．また，G1.1 と G1.2 ，G2.1 と G2.2 のそれぞれで，実験順序による有意差はなかったので，G1 と G2 にそれぞれ統合する．図３，図４に，以上の統合結果の読解時間分布を示す．

２つのプログラムのそれぞれについて，日本語のほうが英

表１．コメントの割合の変化

	ｺﾒﾝﾄ文字数／総文字数	仕様記述／総ｺﾒﾝﾄ	処理記述／総ｺﾒﾝﾄ
不完全日本語版	４５％	６０％	４０％
日本語版	３４％	９１％	９％

表２．プログラムの正しい読解に要した時間．

グループ		被験者	研究分野	課題：数＿日 読み時間(分)	課題：数＿日 回答時間(分)	課題：数＿日 合計(分)	順序	課題：コ＿英 読み時間(分)	課題：コ＿英 回答時間(分)	課題：コ＿英 合計(分)
G1	G1.1	1	AP	7	8	15		8	8	16
		2	S	10	3	13		16	1	17
		3	S	15	*	15		40	1	41
		4	AP	11	2	13	←	20	4	24
		5	AP	7	9	16		18	12	31
		6	AP	22	*	22		30	19	49
		7	S	13	18	18		13	11	24
	G1.2	8	AP	17	3	20		19	12	31
		9	S	10	2	12	→	14	9	23
		10	AP	12	6	18		9	46	55
				課題：コ＿日				課題：数＿英		
G2	G2.1	11	S	11	10	21		30	5	35
		12	S	7	2	9		12	3	15
		13	AP	10	10	20		15	*	15
		14	S	2	9	31	→	29	4	33
		15	AP	16	3	19		30	1	31
		16	AP	20	*	20		23	6	29
	G2.2	17	S	10	5	15		20	20	40
		18	AP	9	2	11		16	2	18
		19	AP	35	*	35	←	44	*	44
		20	S	20	6	26		18	6	24

*: 読み時間と回答時間を分離して記録せず．
研究分野における，AP は application，S は systems software.

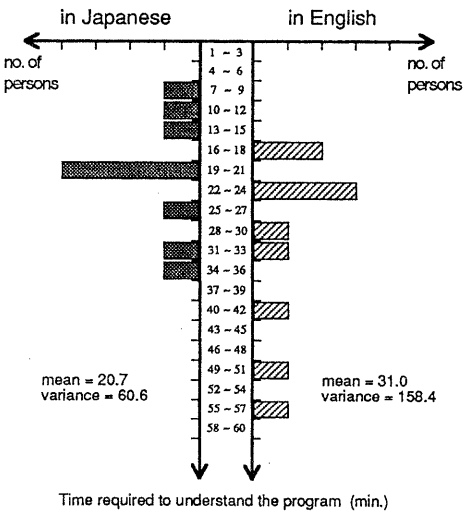

図３　コメント計数の解読時間の分布

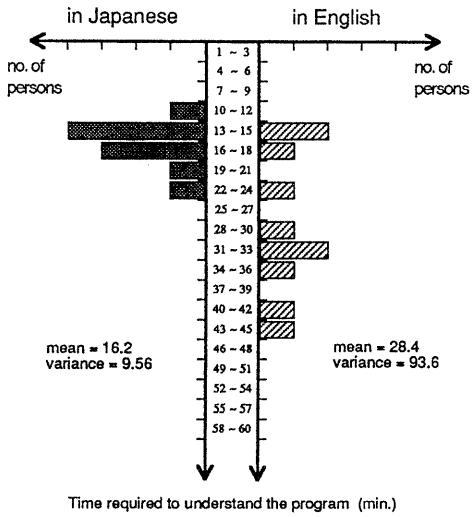

図4　文脈付き数字列検索の解読時間の分布

語より読み易いかどうか検定する目的で，統計量 t を計算し，自由度18 の t 検定を行なった．コメント統計のプログラムについては，t = 2.1 となり，2.5 % の危険率で日本語のほうが早く理解できると言える．数字列検出のプログラムについては，t = 3.6 となり，0.25 % の危険率で日本語のほうが早く理解できると言える．両方の結果から，被験者によらず，日本語プログラムの方が早く理解できる，つまり，読み易いと言える．

4.2 デバッグ効率の評価の試み

プログラムのデバッグし易さに，日本語プログラムと英語プログラムで違いがあるかどうか実験を試みた．デバッグ効率はソフトウェアの品質に影響する重要な問題である．評価実験は，間違いを含んだプログラムリストを提示して間違いを指摘して正しく直してもらい，それに要する時間を計測するというものだった．ところが，紙面上の実験では，日本語，英語によらず正しく虫を見つけ直す人は非常に少なく統計をとれなかった．この解釈として，紙上実験に問題があったと考えている．この形式だと試行錯誤ができないし，実行の確認もできない．しかし現実には，数回の試行を繰り返してデバッグすることに慣れている．だとすれば，実行が確認できるプログラミング環境でユーザの行動を計測する必要がある．

5．考察

5.1 プログラムの国際性

「プログラムは，国際互換性のために英語で書くべき」と言う人がいる．しかし，非英語圏の者にとって，それでソフトウェアの生産性が上がるだろうか．よいソフトウェアが開発されるだろうか．処理やデータに実体を反映した英語名が付けられるか．それを読む労力はどうか．英語を強制することで，他人には意味の分からない英語やローマ字によるプログラムができるだけではないだろうか．結局，そのようなプログラムには，処理を逐一説明するコメントが必要になる．

英語でプログラミングすべきという人は，母国語でコメントを書きことを認めたり，薦めさえする．しかし，コメントは自然言語で記述されるため，コメントをある言語から英語に翻訳するのは自動化し難い．世界中で読まれることを意図されながら，コメントを翻訳しなければ理解し難い“英文字列”プログラムができるだけではないだろうか．それが果して海外でも保守されていくだろうか．

ソフトウェアの国際互換性は重要であるが，この問題はプログラムに限定されるものではない．ソフトウェアには，プログラム以外に仕様，設計，テスト，保守，マニュアルなどの文書が含まれる．プログラムを英語で書けば，国際性を確保できるわけではない．

5.2 プログラミングと国際化の分離

認知心理学では，意識を要する複数の仕事は干渉することが知られている[19]．プログラミングを英語に限ることは，プログラミングと，英語で考える，あるいは，英語に直すことが干渉すると考えられる．だとすれば，中途半端に英語で作るより，母国語で作って変換や翻訳を考えたほうがよいと言える．母国語でよいものができれば，それは翻訳されるということは，日本の古典文学を見れば明かである．ソフトウェアについても，最初から国際互換性を考えて不必要な制限を課すより，まず母国語でよいものをつくって，しかる後に翻訳を考えるべきである．

母国語でプログラミングすれば処理の言い換えのためのコメントはほとんどなくて済む．プログラム自体で可読性が高くなるし，仕様書との対応もよくなる．このプログラムの方が英訳し易いことも分かった．識別子やメッセージの英訳は，機械翻訳ではなくて，変換表で済む．そこで，日本語のトークンと対応する英語の対訳表を用意し，その変換を行なうツールを作成した．このツールを使って，OS/omicron上で開発した日本語プログラムをパーソナル計算機に移植し，利用している．

5.3 日本語プログラム入力の経済性問題とその解決

我々の研究室では，大半の学生が日本語でプログラミングするようになった．その方が，英語で名前を考える必要がないこと，仕様書からの詳細化が円滑なこと，プログラム作成には作成途中のリストを読みことが伴い，日本語の方が思考を発展させやすいこと，保守が容易になること，を理由とし

てあげている．このことは，日本語プログラミングがソフトウェア開発者にとって経済的であることを示している．

一方，英語でコーディングすることを好む学生もいる．これは，英語の方が入力が容易であるという理由による．確かに仮名漢字変換入力はアルファベット入力より手間を要する．もちろん，仮名漢字変換でも略記法を登録して少ない手数で入力効率をあげている学生もいる．

しかし英語入力を好む学生も，一時的に必要とするプログラム以外は，先に述べた日本語プログラムと英語プログラム間の変換ツールを利用して，日本語に変換し保守するようになった．日本語プログラミングを不便にする技術的要因があるとしても，それらを改善するツールを作れば済むことである．

日本語プログラミングとは，何もすべてに日本語を使用すべきということではない．英字はJIS コードに含まれるため，英語で名前付けすることも可能である．ループカウンタなどには，英字で i，j などと書いた方が便利である．

5.4 段階的詳細化

母国語プログラミングの本当の効果は，要求仕様から始まって，幾つかのレベルの設計仕様書を経てプログラムを開発するときに発揮される．人間は母国語で考え，その考えを書き下すものである．文書にするか，プログラムにするかは，それがソフトウェア開発のどの段階にあるかによる．日本語による思考を英語にしなければならないのは，どの段階でも思考を中断させる．

5.5 母国語プログラミングか視覚プログラミングか

言語ではなくて視覚プログラミングに移行するのではないかという意見がある．後者の方が，分かりやすさやデバッグ効率，そして一部の問題では記述力においても優れた面がある．しかし，それだけである程度のソフトウェアができるとは考えにくい．視覚プログラミングで，ほとんどのすべての処理が記述される時代が来ると仮定してみよう．そうは言っても，問題をサブルーチンに分解してプログラムを作る構造化プログラミングやデータ抽象化はともに普遍的であると考えられよう．あるサブルーチンを作ったとき，それに名前付けして再利用するわけだが，これが文字列による名前ではなくて，アイコンまたは図だったとしたら，似たような部品のなかから正しいものを選べるだろうか．また，要求仕様からの段階的詳細化を考えるとき，要求仕様まで図だけで表現できるのだろうか．もし，要求仕様言語を使うなら，プログラムにも言語が残るのではないだろうか．

結局，モジュール間または外部とのリンケージは言語でとることになるのではなかろうか．外部とのリンケージを言語でとるとすれば，母国語である方が覚え易いし，分かり易い．そのためには，サブシステムの母国語化は有効ではなくて，システム全体の母国語化が必要になる．パタンと言語の両方をうまく使いこなすのが人間の知能の本質であろう．したがって，視覚と母国語プログラミングということになるのかもしれない．

5.6 図によるコメント

コメントを図，特に，線図で書くことは，決してメモリを浪費することではなく，プログラムを理解するのにきわめて役立つ．リスティングツールと線を描画するルーチンがフレームメモリを共有すれば，図のコメントを含むプログラムリストを生成することができる．これは，OS/omicron でも実現している．本質的問題は，図を書くとき，どのようにしてユーザフレンドリなインタフェースを提供するかということである．

6．結論

本稿では母国語プログラミングの理念，実現，実践とその効果について述べた．ここでいう“母国語プログラミング”とは，曖昧性を有する自然言語でプログラムを記述するのではなく，プログラミング言語の文法の範囲内で，母国語を最大限に自由に使用することを意味する．

計算機システムの日本語化は，2バイト固定長 JIS コードを内部コードとし，系統的に行った．システム全体の日本語化の中枢は，システム記述言語処理系の日本語化であった．このシステム言語処理系の日本語化についても，ソースプログラムの修正を最小限に抑えて系統的に行なった．

このシステムを用いて相当規模のソフトウェアを開発してきた．日本語プログラミングにより，仕様書からプログラムへの円滑な詳細化が可能になった．仕様書とプログラムの一体化は保守性にも寄与している．また，プログラムの処理内容を日本語のコメントで言い換える必要がなくなり，反対に各関数やモジュールの仕様を記述することを助長している．別に，プログラムの可読性実験を行ない，日本人にとって英語より日本語を用いて記述したプログラムの方が可読性が高いことを検証した．

母国語プログラミングの効果を幾つか述べたが，その本当の効果は，要求仕様から始まって，幾つかのレベルの設計仕様書を経てプログラムを開発するときに発揮される．文書にするか，プログラムにするかは，それがソフトウェア開発のどの段階にあるかによる．日本語による思考を英語にしなければならないのは，どの段階でも思考を中断させる．

ソフトウェアの国際性は重要ではあるが，その問題はプログラミングに限定される訳ではない．なぜなら，ソフトウェアは各種の文書を含むからである．これらまで最初から英語で開発すべきか疑問である．プログラムについては，その識

別子やメッセージを日本語から英語へ，またその逆の変換を行えばよいことである．ソフトウェア全体については，最初から国際互換性を考えて不必要な制限を課すより，まず母国語でよいものを作って，しかる後に翻訳を考えるべきである．

我々は，日本語プログラミングを開始したばかりである．日本人にとって日本語が制限なく使えることの利点は，ソフトウェア工学において，特に，プログラムの可読性，保守性，品質などにおいて，比類のない効果をもたらすものと期待できる．この効果を，実際のソフトウェア工学のなかで評価する必要がある．そのためには，システム側でプログラマの行動を計測し記録していく仕掛が必要である．

謝辞

本稿で述べた内容は，多くの方にその基礎を負っている．高橋延匡教授，並木美太郎助手，また，卒業生，特に，藤森英明，篠田佳博，森岳志，里山元章，屋代寛，田中泰夫，鈴木茂夫，小林伸行，本間正之，堀素史，岡野裕之，下村秀樹，横関隆，鈴木未来子，福島英洋，Rodney G. Webster の各氏に深謝する．

参考文献

[1] 中川，篠田，藤森，高橋：MC68000ユニ&マルチ・プロセッサ・システム用システム記述言語C処理系の開発，情処学計算機システムの制御と評価研資21-7(1983).

[2] 屋代，森，並木，中川，高橋：OS/omicron 用言語Cコンパイラcatの開発 ～VAX/UNIX クロスシステムからの移行～，情処学ソフトウェア工学研資48-1(1986).

[3] 並木，中川，高橋：言語Cコンパイラcatの方式設計，情処学ソフトウェア工学研資 48-2 (1986).

[4] 並木，屋代，田中，篠田，藤森，中川，高橋：OS/omicron 用システム記述言語Ｃ処理系 cat のソフトウェア工学的見地からの方式設計，信学論，J71-D，4，652-660 (1988).

[5] 鈴木，小林，田中，中川，高橋：OS/omicron における日本語プログラミング環境，情処学「コンピュータシステム」シンポジウム，11-18 (1987).

[6] 鈴木，小林，田中，中川，高橋：OS/omicron における日本語プログラミング環境，情処学論，30，1，2-11 (1989).

[7] 並木，関口，里山，屋代，中川，高橋：日本語ワードプロセッサの Programmer's Work Bench 化，情処学第32回全大 5F-5，433-434 (1986).

[8] 並木，関口，鈴木，小林，中川，高橋：OS/o における日本語プログラミング環境と日本語ワードプロセッサの PWB 化，信学論，J71-D，6，994-1003 (1988).

[9] 里山，中川，高橋：OS/omicron における文書出力システム浄書(JOSHO)，情処学第33回全大，4V-12，327-328 (1986).

[10] 里山，中川，高橋：文書の論理構造を備えた日本語清書システム「浄書」の設計と実現，情処学論，30，9，1126-1134 (1989).

[11] 岡野，堀，中川，高橋：多重OS「江戸」の設計と実現，情処学論，30，8，1012-1023 (1989).

[12] T. Souya, E. Hayakawa, M. Honma, H. Fukushima, M. Namiki, N. Takahashi and M. Nakagawa: Programming in a Mother Tongue: Philosophy, Implementation, Practice and Effect, Proc. 15th IEEE COMPSAC., Tokyo, 705-712 (1991).

[13] 島崎真昭：日本語Pascal：パスカル，情処学第23回全大 1H-1，215-216 (1981).

[14] 久野，佐藤，鈴木，中村，二瓶，明石：CLU マシンシステムの開発，情処学オペレーティングシステム研資 33-4 (1986).

[15] 日経エレクトロニクス：特集 国際版OS登場 パソコン・ソフトの国境が消える，日経エレクトロニクス，no. 550，109-131 (1992.3.30).

[16] The Unicode consortium: The Unicode Stadard - Worldwide Character Encoding Version 1.0(1991).

[17] 篠田，藤森，中川，高橋：MC68000マルチプロセッサシステム開発用言語Ｃコンパイラの実現，情処学第28回全大 2H-2，345-346 (1984).

[18] Earley, J. and Sturgis, H.: A Formalism for Translator Interactions, Comm. ACM, 13, 10, 607-617 (1970).

[19] アンダーソン（富田他訳）：認知心理学概論，p.552，誠信出版，東京 (1982).

付録．日本語プログラムの断片

```
/*(4)呼ばれる関数                                              */
/* 圧縮表現化()                                               */
/*(5)注意                                                     */
/*  圧縮表現長(転回部のセグメント長の累積)の格納は,この関数内で */
/* 行う.引き数始めのセグメント番号がdummyならば,転回部の圧縮 */
/* 表現長は0である.                                           */
/*  引き数転回部番号は圧縮表現化()において,代表セグメント格納()*/
/* の引き数代表セグメント番号と同じものである.この関数が呼ばれた後,*/
/* 圧縮表現化()内の代表セグメント番号はインクリメントされる.   */
/* 圧縮表現において,転回部の数は代表セグメントの数に比べて1少ない.*/
/*-------------------------------------*/
FGLOBAL WORD 転回部格納(処理対象,転回部番号,符号付き累積方向差,始めのセグ
        終わりのセグメント番号)
WORD    処理対象;/* @スパン3、@スパン11、@表現
WORD    転回部番号;
WORD    符号付き累積方向差:
WORD    始めのセグ
WORD
              = (転回部番号<<1)+1;
        (転回部データ位置 )= @最大圧縮表現数){
            return(@オーバーフロー);
        )
        if(符号付き累積方向差 ) 0)
            方向種 = @右転回;
        else
            方向種 = @左転回;
        if(処理対象 == @スパン3){
            _3点近似圧縮表現列[現在画番号][転回部データ位置].転回.種類
                = (BYTE)方向種;
            _3点近似圧縮表現列[現在画番号][転回部データ位置].転回.累積方向
                = (BYTE)(abs(符号付き累積方向差));
            圧縮表現長 = 0;
            _PH2_3点近似セグメント列P += 始めのセグメント番号;
            for(
                ワーク1 = 始めのセグメント番号;
```