Lisp の ISO 標準規格 ISLISP の処理系の研究開発

A Development of a Portable Processor for ISO Lisp Standard ISLISP

湯浅 太一[*1]
yuasa@kuis.kyoto-u.ac.jp

梅村 恭司[*2]
umemura@tutics.tut.ac.jp

長坂 篤[*3]
nagasaka@okilab.oki.co.jp

*1 京都大学大学院情報学研究科 *2 豊橋技術科学大学情報工学系 *3 沖電気工業株式会社研究開発本部

ISLISP はプログラミング言語 Lisp の ISO 標準規格であり，伊藤，湯浅，梅村らによる KL をベースとして開発された．本研究開発では，この ISLISP の普及を目的として，移植性が高くかつ高速な ISLISP 処理系を開発した．本処理系では、移植性と開発の容易さから仮想マシン方式を採用し、ISLISP プログラムは Byte Code と呼ばれるこの仮想マシンの命令語に変換され、Byte Code インタープリタによって解釈実行される．

仮想マシンコード方式は、高速な言語処理系の実装に適した方式ではないが、本処理系では Lisp オブジェクトの構成や Byte Code インタプリタの高速化によって，十分に高速な処理系を実現した．

1. はじめに

ISLISP は、1997 年に制定された、プログラミング言語 Lisp の ISO 標準規格であり、ISLISP の核言語として日本で開発された KL(Kernel Language)をベースとしてしている．

本研究では、この ISLISP の普及を目的として、移植性に優れ、かつ ISLISP の特徴である高い実行効率を生かした高速な処理系の開発を行った．

本稿では、ISLISP の概要、本処理系の構成を Lisp オブジェクトの構成を中心に説明し、最後に性能評価結果について報告する．

2. ISLISP 概要

2.1 ISLISP 標準化の経緯

Lisp は 1950 年代末に，人工知能研究におけるプログラミングシステムを目標として、MIT の J. McCarthy 等によって開発された，ラムダ計算を理論的基礎とする関数型プログラミング言語である．LISP は言語の持つ拡張の容易さや他の言語にはない機能によって，主に大学や研究機関を中心として MacLisp や Interlisp，Lisp Machine Lisp，Scheme 等の多くの方言が開発されてきた．このように多くの Lisp 方言が開発されると、プログラム間の互換性の問題が発生し，当然ながら ISLISP 以前にも Lisp の標準化の活動があった．最初の試みは、Utah 大学 による Standard LISP であったが、これは標準化の観点からは大きな成果はなかった．

1970 年代の AI 研究は，1980 年代に入るとエキスパートシステムや知識工学に代表される実用化研究に移り、Lisp を産業界における実用的言語とするために標準化の必要が認識されていた． ARPA は 1981 年春、Lisp Community Meeting を開催し， ここで Lisp の標準規格を開発することが Lisp ベンダおよび研究者間で合意され，LispMachine Lisp を含む MacLisp 系の Lisp を統合する Common Lisp 開発が始まった．Common Lisp は Guy Steele Jr. によってまとめられ，1984 年“Common Lisp the Language" (CLtL1)[2]として出版された．Common Lisp は，MacLisp 系の Lisp 及び Scheme を統合し， Lexical Scoping, 多値(Multiple Values), Value cell と function cell の分離 (Lisp-2)， Structure (DEFSTRUCT)，汎変数 SETF，IEEE 準拠の浮動少数演算，Lisp Machine Lisp 風の強力なラムダリスト等の機能を持っていた. さらに, 国際標準を目指して, ANSI X3J13 によって CLtL1 の言語仕様の矛盾点の解消，この頃までに大きな技術的発展を遂げていたオブジェクト指向機能や、例外処理等の新しい機能の導入等が行われ，CLtL2 を経て 1992 年 ANSI Common Lisp が制定された．1984 年の CLtL1 の発表以来，折りからの AI ブームもあって多くのベンダが Common Lisp の製品化を行ない，Common Lisp の産業界の標準としての位置は確立された．

†本研究は情報処理振興事業協会「独創的先進的情報技術に係わる研究開発」の一環として行われたものである。

2.2 ISLISP 標準化

一方，Common Lisp の発表以来，その言語仕様の巨大さから，実行効率や学習・利用の面からの問題が指摘されてきた． Common Lisp の抱えるこのような問題を解決し，コンパクトで効率が高く，かつ使いやすい Lisp 言語を目標として， 1987 年に SC22 に WG16 が設置されて ISLISP の標準化が開始された．

ISLISP の目的は、一貫した言語仕様をもち、産業界での使用に耐える得る品質を持った、Common Lisp のサブセットを開発することであり、Common Lisp の巨大さに対して、核言語をベースとする Layerd アプローチをとった．このような核言語の候補として、当初 Eulisp と Common Lisp が候補であった。最終的に、ISLISP のベースドキュメントとして，LISP の核部分に対して伊藤，湯浅，梅村等の KL (Kernel LISP)[3]が，オブジェクト指向機能に対して Common LISP の CLOS(Common Lisp Object System) をベースとした ILOS(ISLISP Object System)が採用され、1992 年 ISLISP の最初のドラフトが設計され、1997 年に ISLISP は IS 規格として制定された[1]．ISLISP は現在 情報処理学会 ISLISP JIS 原案作成委員会において JIS 化作業が進められている．

2.3 ISLISP の特徴

ISLISP は，当初 Common LISP のサブセットを目標として開発され，Common LISP の基本的な言語仕様を受け継ぐと共に，Scheme を意識したコンパクトな仕様や構文の影響を受けている． LISP としての ISLISP は，コンパクトな仕様と実行効率の高さ，効率の高いオブジェクトシステム，言語仕様と処理系仕様の分離，データ型のオブジェクトシステムへの統合等の特徴を持つ．仕様の検討が行なわれながら，今回の ISLISP 規格に含まれなかった機能として，パッケージあるいはモジュールがある．実用的なソフトウェアの開発では，名前の衝突を解決するこれらの機能は不可欠であり，今後の課題である

3. ISLISP 処理系の構成

本研究の目的は、ISLISP の普及を促進するために移植性が高く、かつ実用的な性能をもった ISLISP 処理系を早期に開発することである．

移植の容易さと性能の両立は一般に両立が困難な目標である．これまでに開発された移植性と高速性を両立させた Lisp 処理系には以下がある．

(1) PSL(Portable Standard LISP)[5]

(2) Kyoto Common Lisp(KCL)[6]

(3) Tachyon Common Lisp[7]

PSL は， SYSLISP で記述されたコンパイラ(PCL)が，Standard Lisp を抽象的なアセンブラにコンパイルし，その後でアセンブラマクロ(c-macro)がパターンマッチによってマシン依存なコードに変換する Retargetable なコンパイラである．この方式は魅力的ではあるが，マシン毎にマクロを用意する必要があるのが欠点である．

KCL は Common Lisp を等価な C プログラムに変換する処理系であり，マシン固有なコード生成処理を対象マシンの C 処理系に行なわせることによってマシン独立性を実現している．しかしながら，最近のユーザ環境，特に PC では，C 処理系を備えていない場合が一般的であること，C 処理系の互換性及び起動に要する時間に問題があることから,本処理系では採用しなかった．

Tachyon Common Lisp は，RISC マシンを対象とする商用 Common Lisp 処理系として開発された高速処理系である．仮想的な RISC マシンとその抽象アセンブラを想定し，この上に処理系が開発されている．抽象アセンブラは S 式による機械独立なシンタックスとマクロ機能を持っているためインタプリタの移植は容易であり，またコンパイラの移植も 2 人月程度で可能である．しかしながら，多数のレジスタを備える RISC マシンを対象とし，実行効率の観点からこれらのレジスタを直接使用しているため，今回の重要なターゲットである Pentium プロセッサへの適用には大幅な設計変更が必要である．

以上から上記の 3 つの方式はいずれも問題があり、我々は限られた時間内での移植の容易さを優先して，仮想マシン(Byte Code Machine)方式を採用した．

3.1 仮想マシンの仕様

仮想マシンは, スタックマシンをベースとして, Lisp プログラムの実行に最適化した Byte Code を命令語として持つ．ISLISP プログラムは，Byte Code Compiler によって等価な Byte Code プログラムに変換され，Byte Code プログラムは Byte Code Interpreter によって解釈実行される．本処理系は Byte Code による実行のほかに，メモリ上に読み込まれた ISLISP プログラムを直接解釈実行する Interpreter も実装している．

図 1 に ISLISP 処理系の構成を示す．

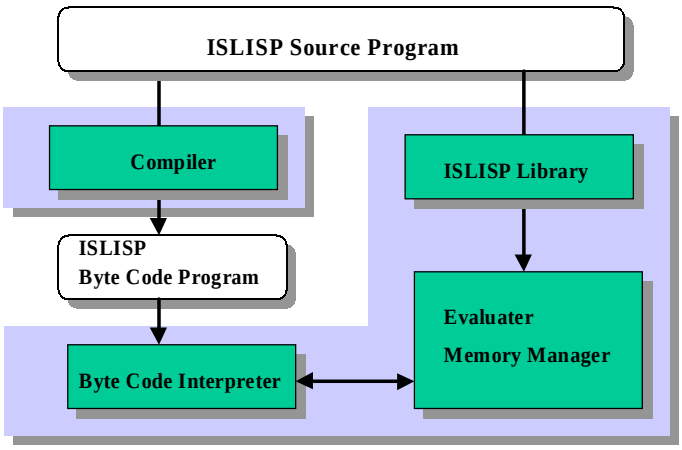

図 1.　ISLISP 処理系の構成

以下に本処理系の Byte Code 仕様を示す.

- Byte Code は，1 あるいは 2 Byte の Opcode と 1, 2 あるいは 4Byte の Operand を持つ
- Byte Code は，スタックマシン上で Lisp プログラムを効率よく実行可能であるように規定されている．表 1. に Byte Code の一覧を示す.

4. Lisp オブジェクトとメモリ管理

4.1 Lisp オブジェクトの構成

Lisp オブジェクトの構成は，(1) 高速な実行が可能であること，(2) データ領域に制限が無いこと，を目標として設計された.

LISPオブジェクトは32ビット長のデータであり，アドレス部とタグ部からなる．タグは8ビットまたは 3ビットであり，下位に割り当てられる．また，最下位ビットはGC用に使用する． タグ部を下位に割り当てることにより，高速なメモリアクセスが可能となる.

(1) CONS, SYMBOL

これらのデータは，8バイト境界に配置したメモリ領域に格納している．このため，アドレスの下位3ビットは必ず0になる．これらのタグは，3ビット(CONSは000，SYMBOLは100)になっているので，LISPオブジェクト全体が実体への直接のアドレスとなり，高速にアクセスすることができる.

(2) 即値(固定長整数，文字)

これらのデータは，アドレス部に値をそのまま格納する．固定長整数は24ビット整数であり，アドレス部にデータの値そのものを格納する. 本処理系では，日本語などの2バイト文字を考慮して，すべての文字を16ビットで扱う．アドレス部の上位16ビットに，文字コードを格納する.

(3) 他のタイプ

上記以外のタイプのデータは，アドレス部に，HEADER メモリのオフセット値を格納する．GC 時に圧縮または複写によりアドレスが移動しても LISP オブジェクトのアドレス部は固定になる．アドレスでなくオフセット値を格納することにより，LISP オブジェクトの値が変化しないので，高速な処理ができる.

表 1　Byte Code

分　類	機　能
スタック操作	スタックポインタを移動
	SP から Offset 位置の内容を参照
変数アクセス	局所変数へのアクセス
	クローズ情報中の変数の参照
	クローズされたスタック上の変数の参照
	グローバル変数の参照
	dynamic 変数の参照
Lisp Obj 即値	1 Byte 文字(1Byte operand)
	2 Byte 文字(2Byte operand)
	2Byte Lisp object(2Byte operand)
	4Byte Lisp object(4B operand)
	特別な Lisp object(operand なし、NIL、UNDEF 等)
関数呼出し	一般的な関数呼出し
	FUNCALL を使用したラムダ式呼出し
	最適化用のシステム関数の直接呼出し
分岐	無条件分岐
	条件が真ならば分岐
	条件が偽ならば分岐
	条件が真ならば値をそのままで分岐
	条件が偽ならば値をそのままで分岐
特殊形式	FUNCTION
	LAMBDA
	DYNAMIC フレームの生成、解放
	データ比較
	BLOCK フレームの生成
	RETURN-FROM
	CATCH フレームの生成
	THROW
	TAGBODY フレームの生成
	GO
	CLASS
	ASSURE
	CONVERT
	STANDARD-STREAM
	IGNORE－ERRORS
	WITH-HANDLER
	UNWIND=PROTECT
	Clean Up フォームの実行、解放

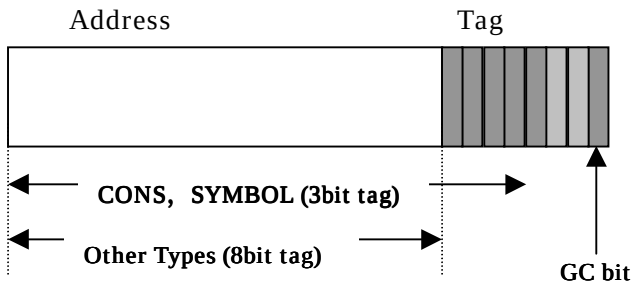

図 2. LISP オブジェクト

4.2.メモリ管理

4.2.1 メモリブロック

本処理系では，さまざまなデータを格納するために，独自のメモリを使用しており，これをLISPメモリと呼ぶ．メモリブロックは，まとまった大きさで確保されたLISPメモリと，その先頭に付加されたヘッダ情報から構成される．本処理系では，表に示す5種類のLISPメモリに分け，それぞれの領域を分割して管理している．

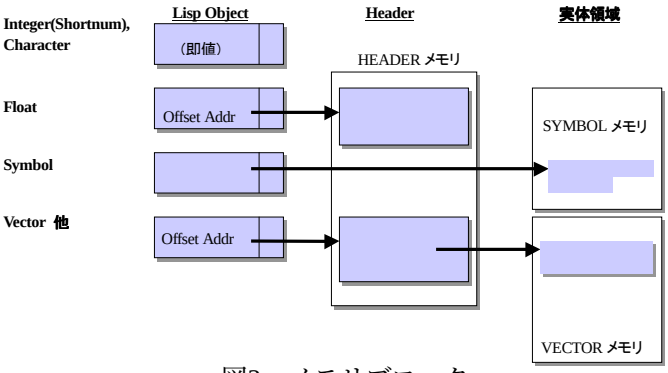

図3　メモリブロック

表2　LISPメモリ

名称	拡張	アライメント
CONS	追加	8バイト
SYMBOL	追加	8バイト
HEADER	伸長	8バイト
VECTOR	伸長	4バイト
STACK	伸長	4バイト

メモリの拡張を行うとき，CONS, SYMBOL はメモリブロックの追加を行うが，他のメモリは，全体が連続した領域である必要があるので，メモリの伸長を行う．メモリの伸長は，現在よりも，より大きなメモリブロックを獲得し、そこに現在の内容をコピーすることによって行う．

以下に，各LISPメモリについて述べる．

(1) CONSメモリ

CONSデータを格納する．CAR部とCDR部から構成され，それぞれに任意のLISPオブジェクトを格納する．

(2) SYMBOLメモリ

SYMBOLデータを格納する．シンボル名や，シンボルの値などを格納する．

(3) HEADERメモリ

CONSおよびSYMBOL以外の副構造を持つ LISPオブジェクトは，アドレス部に HEADERメモリへのオフセット値を格納する．このため，HEADERメモリ全体は常に連続な領域となっている．

(4) VECTORメモリ

VECTORメモリには，文字列やベクタなどの可変長データを格納する．VECTORメモリが使用される場合，その領域へのポインタは，常にHEADERメモリに格納する．

(5)　STACKメモリ

LISPスタックとして使用する．

4.3　ガーベッジコレクション(GC)

本処理系では，一括型のGCである，マーク/スイープ方式を採用している．本処理系では，マーク/スイープ方式による一括型のGCを採用している．GCによって，CONSメモリ，SYMBOLメモリ，HEADERメモリの回収とVECTORメモリの圧縮が行われる．

このとき，CONSメモリ，SYMBOLメモリは移動せず，また，HEADERメモリのオフセット値は変化しない．これによって，GCの前後で，LISPオブジェクトの値は変化しないので，GCの前後で参照するLISPオブジェクトのアドレスの変化を気にしなくてよいので，高速に処理が行われる．

このとき，CONSメモリ，SYMBOLメモリは移動せず，また，HEADERメモリは，常に全体が連続した領域であるため，オフセット値は変化しない．即ち，GCの前後でLISPオブジェクトの値は変化しない．よって，あるLISPオブジェクトをGCの前後で参照するとき，GCの後で，LISPオブジェクトの値が変化したかどうかのチェックをせずに済むため，高速に処理が行われる．

5. ISLISP オブジェクトシステム

ISLISPのオブジェクトシステムは，COMMON LISPのオブジェクトシステムであるCLOS(Common Lisp Object System)のサブセットをベースとしている．ISLISPのオブジェクトシステムは、CLOSと比較して機能が大幅に制限されており効率的な処理系の実装が可能となる．また，ISLISPの仕様は処理系依存になっている部分が多いため，その部分をどのような仕様にするかによって実装に大きな影響を与える．

ここでは，本処理系におけるオブジェクトシステムの実装について述べる．特に効率的なクラスの再定義のために導入した，古いインスタンスの管理用のクラスやウィークポインタについて述べる．

5.1 ISLISP のオブジェクトシステム

ISLISPのオブジェクトシステムは，COMMON LISPのオブジェクトシステムであるCLOSのサブセットになっており，包括関数(CLOSでは総称関数)を動作の基本としたオブジェクトシステムであり，完全なオブジェクトシステムである．

ISLISPはほとんどのメタオブジェクトプロトコルが処理系依存になっているため，依存部をうまく定義することによりCLOSと比較してコンパクトになり，効率的な処理系の実装が可能となる．またISLISPの仕様では，クラスの再定義などの動的変更については処理系依存になっている．

本処理系では，ISLISPの言語仕様の処理系依存の主な部分を以下のように定義した．

(1) ユーザ定義クラスのメタクラスは、
<StandardClass> のみとする
(2) ユーザ定義包括関数のメタクラスは，
<StandardGenericFunction> のみ
(3) クラスの再定義を可能にする
(4) 関数，メソッドの再定義を可能にする

(1)と(2)により，メタクラスの新規作成を制限することにより，メタクラスのオーバヘッドをなくしている．これは，通常のLISPプログラムにおいては妥当な制限になっている．

このように定義することにより，コンパクトな処理系の実装ができる．我々はオブジェクトシステムを含む処理系のすべてをC言語で記述することにより，高速な処理系を作成した．

また，ISLISPは動的変更についての仕様が処理系依存になっているが，クラスの再定義等は必須な仕様と考え，(3), (4)のように再定義可能にした．

5.2 クラスの再定義

本処理系ではクラスの再定義を可能とした．ここでは，効率の良いクラスの再定義の実装について述べる．

クラスが再定義されたとき，そのクラスのすべてのインスタンスとそのクラスのすべてのサブクラスのすべてのインスタンスの変更を行う必要がある．この変更の方法には以下のものがある．

- 一括型

  クラスの再定義が行われたときに，そのクラスのインスタンスとそのクラスの全てのサブクラスのインスタンスの変更を行う. この方式では，再定義の処理には時間がかかるが，インスタンスの操作は高速である．

- 逐次型

  変更されるクラスやインスタンスの操作が行われたときに変更を行う．この方式では，再定義の処理は高速であるが，インスタンスの操作は遅い．

本処理系では，実行時(インスタンスアクセス時)の高速化のために一括型を採用した．また，再定義の処理を高速にするために，後述のウィークポインタとエラー処理用クラスを導入した．

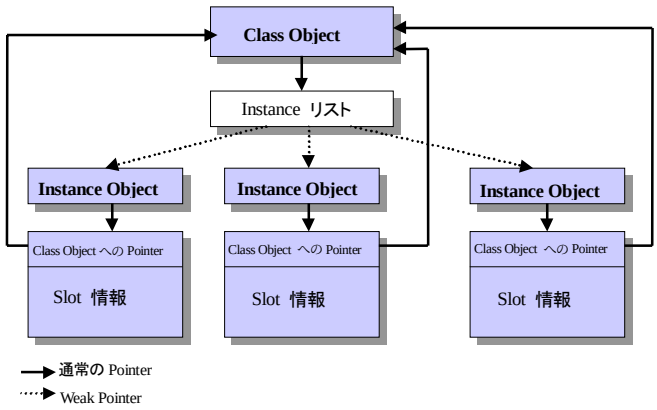

図4 オブジェクトシステムの実装

5.3 エラー処理用クラス <Invalid>

前述したように，本処理系ではクラスの再定義を可能にする仕様にした．クラスの再定義を可能にすることにより，以前のクラス定義で生成されたインスタンスをどのように扱うかを決めなければならない．

本処理系では，以前のクラス定義で生成されたインスタンスに対する操作はエラーを通知するようにした．そのために，特別なエラー処理用のクラス<Invalid>を導入した．

本処理系では，クラスが再定義されたとき，以前のクラス定義で生成されたインスタンスはクラス<Invalid>のインスタンスに変換するようにした．そして，クラス<Invalid>のインスタンスに対する操作はエラーを通知するようにした．図3 にクラス再定義時のエラー通知の例を示す．

```
;;; クラスcircleを定義
ISLisp>(defclass circle (figure)
          ((radius :accessor circle-radius
                   :initarg radius)))
CIRCLE
;;; クラスcircleのインスタンスを生成し，
;;; グローバル変数fig1に設定する．
ISLisp>(defglobal fig1 (create (class circle) 'radius 1))
FIG1
;;; fig1の半径を調べる
ISLisp>(circle-radius fig1)
1
;;; クラスcircleを再定義する．
;;; ここで，以前の定義で生成されたインスタンスfig1は
;;; クラス<invalid>のインスタンスに変換される．
ISLisp>(defclass circle (figure)
          ((radius :accessor circle-radius
                   :initarg radius
                   :initform 1)))
CIRCLE
;;; fig1の半径を調べる
;;; しかし，fig1はクラス<invalid>のインスタンスなので
;;; これに対する操作はエラーが通知される．
ISLisp>(circle-radius fig1)
> Error at CIRCLE-RADIUS
> No applicable method:
```

図5 クラス<Invalid>のインスタンスの操作例

クラス<Invalid>の導入により，インスタンスのアクセス時に古いインスタンスであるかどうかのチェックが不要となり，操作が高速に行うことができる．

5.4 ウィークポインタ

ウィークポインタのみで参照されているオブジェクトはゴミになり，ガーベッジコレクション(ゴミ集め)により領域の再利用ができる

本処理系では，全てのインスタンスをクラスからのウィークポインタで管理する．これによりクラスの再定義が行われたときは，クラスからウィークポインタをたどることにより，そのクラスに属するすべてのインスタンスをクラス<Invalid>に変更する(一括型変更)操作を高速に行うことができる．

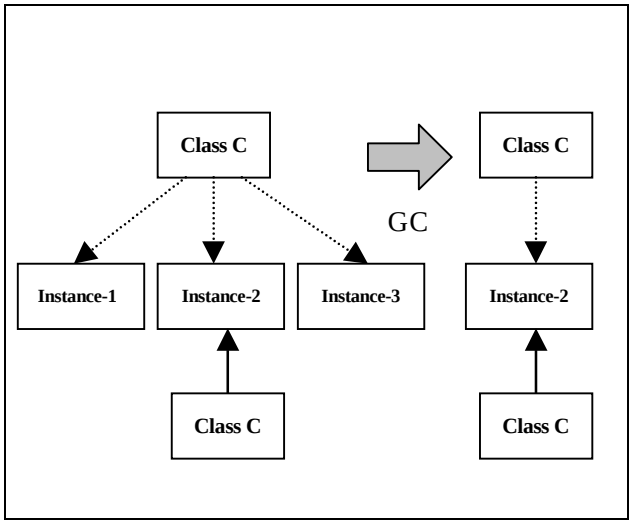

図6 Weak Pointer

このウィークポインタを通常のポインタで管理すると，インスタンスは不要になってもゴミにならないが，ウィークポインタを導入すると使われていないインスタンスはゴミになって回収され，効率的なメモリ利用が行われる．

6. 処理系の実装と性能評価

本処理系は，Windows95/NT, UNIX などの代表的なOSで動作するように，インタプリタの大部分を ANSI C で記述している．GUIは持たずに基本的なコマンドインタフェースのみを持つことにより，また，LISPオブジェクトの構造やアクセス方法も高速性を失うことなく各種CPUで利用できることにより，高い移植性を実現している．

ただし，浮動小数点数演算のオーバーフローとアンダーフローをチェックする部分は一部アセンブラで記述している．

本処理系の性能を評価するために，Gabriel Benchmarkの実行速度を表に示す．測定で使用したマシンは Pentium 133MHzをCPUとするWindows95 PCである．比較のために，Christian JullienによるISLISP処理系であるOpenLispの値を示す．

表3　Gabriel Benchmark の実行速度

Function	Interpreter	Compiler	OpenLisp
Tak	0.840	0.233	0.270
ctak	1.373	0.412	0.570
stak	0.975	0.316	0.370
takl	5.507	1.373	1.930
takr	0.865	0.226	0.370
derivative	1.415	0.617	1.040
dderivative	1.539	0.627	1.100
div2(iterarive)	1.648	0.452	0.550
div2(recursive)	1.934	0.523	0.650
browse	19.446	4.147	5.560
destructive	2.019	0.687	0.970
init-traverse	12.812	4.682	5.400
run-traverse	90.848	25.381	39.050
fft	1.044	0.206	0.410
puzzle	17.276	4.312	6.630
triangle	195.798	70.753	80.190

(単位:　秒)

この表から，本処理系が Open Lisp に対して 1.42 倍高速であることがわかる．また，Open Lisp は以下の問題があり，処理系自体の性能としては本処理系がさらに高速であると考えられる．

- bignum をサポートしていない
- 一部のシステム関数や特殊形式では，引数の個数のチェックを行っていない

7. ISLISP 検証システム

ISLISP 検証システムは、ISLISP 処理系が ISLISP 規格に準拠していることを検証するシステムである．本研究開発では、ISLISP 規格の解釈を規定し、ISLISP 言語仕様書を補足する観点から、ISLISP 検証システムを開発した．

本検証システムは以下の機能を持っている．

- 検証データを用意することにより、自動的に動作確認が可能
- エラーが発生する場合も、ISLISP 規格で規定する正しいエラーが発生しているかを確認可能

検証データは次のいずれかが記述可能である．

(1) 正常な実行
(2) エラー
(3) 単なる実行(前準備)
(4) 述語(データ型チェック)
(5) 引数のデータ型チェック
(6) 引数の個数チェック
(7) 検証システム以外で使用する情報

図 7 に ISLISP 検証システムの構成を示す。検証システムは、検証データ及び検証システムからなり、検証システムは、検証対象の ISLISP 処理系に対して、検証データのフォームを与え、評価されたフォームの結果と検証データの規定する値とを比較することによって、ISLISP 言語仕様への合致性を検証する。

検証データは、ISLISP 言語仕様の補足及び解釈の統一を目的として作成され、現在 16000 項目のデータからなる。

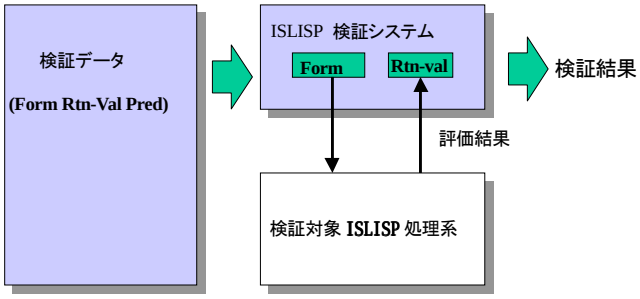

図 7　ISLISP 検証システム

8. おわりに

日本案をベースとして開発された LISP の ISO 国際規格である ISLISP の普及を目的として、移植性が高くかつ高速な ISLISP 処理系を開発した．また、ISLISP 処理系の ISO 規格への合致性を検証する ISLISP 検証システムを開発した。

処理系は移植を容易にするために Byte Code マシン方式を採用している．Byte Code 方式は高速な処理系の方式として最適ではないが、Lisp オブジェクトの構成等により、他の ISLISP 処理系と比較して十分高速な ISLISP 処理系を実現した．本処理系は現在、UNIX(Linux 、 BSD Unix 、 Solaris 、 HP-UX) 、Windows95、WindowsNT で動作している．

参考文献

[1] ISO/IEC 13816:1997(E), Information Technology-Programming Languages, Their Environments and System Software Interfaces -Programming Language ISLISP, 1997

[2] Guy L. Steele Jr., “Common LISP the Language, 2nd edition,” Digital Press, 1990

[3] 伊藤貴康，LISP 言語国際標準化と日本の貢献，情報処理 38 巻 10 号, 1997 年 10 月

[4] プログラム言語 ISLISP JIS X 3012，日本規格協会， 1998 年

[5] Univ.Utah, “The Portable Standard LISP Users Manual,” Tech Rep.TR-10, 1982

[6] 湯浅太一, 萩谷昌己: Kyoto Common Lisp Report, 帝国印刷, 1985.

[7] Atsushi Nagasaka et al., “Tachyon Common Lisp: An Efficient and Portable Implementation of CLtL2,” ACM Lisp and Functional Programming Symposium, 1992

[8] 新谷 義弘他，ISO 規格 ISLISP 処理系の開発，情報処理学会第 56 回全国大会予稿集 5E-06，1998 年 3 月

[9] 各務 寛之他，ISO 規格 ISLISP 処理系の実装方式情報処理学会第 56 回全国大会予稿集 5E-07，1998 年 3 月

[10]各務 寛之他、ISO 規格 ISLISP 処理系におけるオブジェクトシステムの実装について，情報処理学会第 56 回全国大会予稿集、1998 年 3 月