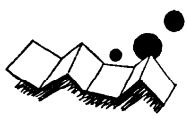
解　説



Lispシステムにおけるデバッギング・ツール†

黒　川　利　明††

1. はじめに

デバッグ（虫取り）という行為を大抵のプログラマは，あきあきする程経験しているに違いない．したがって，虫取りについての知識は皆が一応持っているし，虫取りソフト* を用いた経験も少しはあろう．これに反して Lisp については，その名前を知っていても，Lisp システムの内容を良く知っている人は割り合い少ないだろうと思われる．

この解説の狙いは，良く知られた（と思われている）虫取りという行為を Lisp システムの観点から見直そうというものである．したがって，読者としては一応の Lisp の知識をもっているが，Lisp システムについての経験は余り無い人々を考えている．Lisp を全く知らない人々は，この稿を読む前（もしくは後ほど）に，Lisp についての入門書[1),16),21),26)]を読まれることをお勧めする．

さて，「この良く知られていない Lisp システムから見たデバッギングという事柄には，珍らしいという事以上に何か意味があるのか」と疑う人がいたとしても不思議はない．いわゆる実用上の観点から見るなら，Cobol での虫取りソフトとか Fortran での虫取りソフトの方が有用であって，Lisp の虫取りソフトなどは役に立つまいと思う読者がいても驚くには当らない．

しかしながら，わざわざ「解説」の一環として Lisp システムの虫取りソフトを取り上げるのは，これが Lisp ユーザだけに役立つのではなく，Lisp ユーザ以外の人々にとって有用であると信じられたからであろう．その理由は Lisp のもつ**プログラミング・システム**という性格にある．すなわち，単にプログラムを記述するだけではなく，修正し，テストし，効率化するといったプログラム作成上の全行為を結合したシステムという Lisp の今日的性質が，Lisp ユーザだけでなくすべての読者にとって参考になると思われるからである．

さて，プログラミング・システムという概念は，今日では，ソフトウェア工学の一環として，漸くその位置を確保したようである．今年のソフトウェア工学シンポジウムにおいては和田[32)]が，ACM の Computing Surveys においては Sandewall[25)]が，それぞれプログラミング・システムについて述べている．この両者の論文が共に Lisp プログラミング・システムを取り上げているのは偶然の一致ではない．プログラミング・システムとしてある程度完成されたものは，現在のところ Lisp に基づいたものしか存在しないからである．ただし，これは Lisp でなければプログラミング・システムが作成しえないということを意味するものではない．プログラミング・システムには何が必要かという具体的な要件に関しては 25) が詳しい．

Lisp システムの虫取りソフトは，このプログラミング・システムの重要な構成要素の一つとして位置づけられる．実際それは単なる道具の寄せ集めではなく，プログラミング・システムのさまざまな要素と関わりをもつ有機体になっている．その個々の内容に関しては，第5章で具体的に述べることにしたい．

次の第2章では，虫取りを再度システム的見地から

† Debugging Tools in Lisp System, by Toshiaki KUROKAWA (Toshiba R & D Center).

†† 東京芝浦電気(株)総合研究所

* debuggingtool を日本語にどう訳せば良いかという事を竹内郁雄氏（電電公社武蔵野通研）と議論した．候補としては，デバッギング・ツール，デバッグ・ツール，デバッガ，デバッグ道具，デバッグ用具，虫取り道具，虫取り用具が掲げられた．英語を素直に仮名に直したものとしては，最初のデバッギング・ツールが最も望ましい．デバッグ・ツールはいかにも和製英語的で気に入らない．デバッガは意味的に少し異なる．tool を道具と訳すのは定訳ではあるが，何となくしっくりこない．tool と instrument の相違を考えるに，tool の方が手工具的色彩が強いようである．名案の浮かばないままに，ここでの tool が実は software を指しているので，虫取りソフトと訳してみた．言葉の坐りもそう悪く無く，実体をかなり良く反映しているのではないかと思う．
難点は和製英語的臭さが残ることと「ソフト」の意味を岩波国語辞典（1970年版）で引くとソフト帽，ソフトドリンク，ソフトクリームの三つの略としてしか載っていない点である．software tool が日本語としてどう訳されるのかと関係しているであろう．本文中ではこの「虫取りソフト」で debugging tool を意味している．debugger は debugging tool とはニュアンスが異なるのであるが，本文中では同じく「虫取りソフト」として訳している．

検討したい．第3章では，プログラミング・システムの概要を述べることにする．第4章では，特に Lisp プログラミング・システムの成り立ちと特徴を述べて，Lisp に馴染みの薄いユーザへの手助けとしたい．

第5章では，本解説の主目的である個々の Lisp システムにおける虫取りソフトの紹介を行う．第6章では，虫取りソフトを分類してまとめる．第7章では，前章のまとめをうけて，将来の虫取りソフトについて筆者の個人的観測を述べようと思う．第8章では，他のプログラム言語での虫取りソフトを充実する上で参考になると思われる点をまとめたい．

2.「虫取り」再考

虫取りは，我々プログラマには日常茶飯事である．(少なくとも虫取りの必要のない超プログラマを除いては！) しかしながら，プログラムの経験の無い一般の人々にとっては，あまりわかり易い行為ではない．実際「何故そんなに虫が出るのか？ プログラマはそんなに不注意なのか？」というような問いかけが良く聞かれるのである．

そこで虫の出る原因として，文法の問題とか，データ構造上の問題だとか，インタフェース上の問題とかを数え上げるのであるけれども，それらがプログラマの不注意に帰せられるものなのかどうかが結局は問題となる．実際，心理学的な観点から言えば (22) の 161 頁参照)，発生する虫の個数は普通に人間の不注意によって引きおこされるにしては，大きすぎるらしい．したがって，虫がこのように大々的に発生するのは，我々が現在用いている道具や方法が不完全なのだという説が生じる．例えば Dijkstra は，彼の発明した方法を用いて開発したプログラムは，(虫が出る事がないので) テストする必要が無いというようなことを言っている[6]．しかしながら，このような方法は，未だ充分に普及していないし，実用的であるかどうかには疑いが持たれている．要するに「完全な」方法や道具というものは，我々が現実に使用することのできるレベルには至っていない (もしくは至りえない？) のである．

虫の発生に対するもう一つの理由としては，プログラムの実際の行動と期待されている行動との間に不一致が存在することが挙げられる．この不一致は，プログラマだけの責任ではない．むしろ，プログラム発注者 (要求者) の側に責任のあることの方が多い．よくあることだが，プログラムがどう動くべきかという事柄を，プログラムが出来て実際に稼動し始めるまでは，良く認識していない人が多いのである．したがって，ソフトウェア工学においては，プログラム作成の前段階である要求定義や仕様定義の方法に皆が注目しているのである．

さて，今述べたような原因が将来除去されたと仮定した場合には，我々が「虫取りという処理が昔は行われたことがあったらしい」というように話すことの出来る日が来るのであろうか？ この答は残念ながら「否」である．(もし，「然り」と答える読者がおられたら，是非誌上討論したいものと考えています)．

その理由は第1に，優れた方法論や道具を用いたとしても，不注意な虫というものは人間が不注意である限りは絶望的に防げないものだということである．第2の理由は，プログラムは商品として，あるいは道具としてさまざまな人々に使用されるものだということである．したがって，プログラムに対する要求は，どうしても多様になり，必ずプログラム作成時と異なる要求が出されるようになるからである．たとえユーザが1人しかいない場合でも，彼の要求は時間と共に変化せざるを得ないのであるし，プログラムの環境も (ハード，OS，ユーティリティ等を含めて) 必ず変化するものなのである．

この第2の理由は，厳密にいえばプログラムの修正 (adaptation) であって，虫取りとは呼べないかも知れない．しかし，定義はともあれ，その時行われる内容は，今ここで述べてきた虫取りと全く同一である (広い意味で虫取りに含めることができると思う)．

最後の理由としては，図-1のようにソフトウェア作成過程を考えると，たとえ虫取りの行為が不要になったにせよ，どこかでこのプログラムが，ユーザの要求を満たしていることを検証する必要があるということ

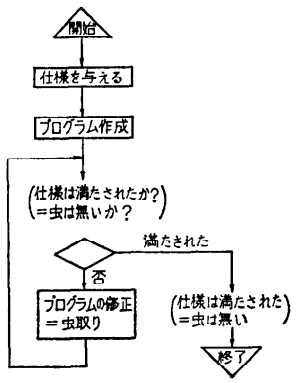


図-1 ソフトウェア作成過程の一例

である．虫の発生を許さない完全なプログラム作成システムが出来上ったとしても，そのシステムが妥当かどうかは，作成されたプログラムがユーザ要求を満しているかどうかのチェックを何らかの形式で行わないとわからない．このチェック機能は，一般の虫取り過程の一部分となっているのである．

以上の理由から虫取りは，プログラムのある限り存在する行為であると言えよう．したがって，プログラマに対しては「虫を作らないように！」というばかりではなく，効果的な虫取りソフトを与えねばならない．実際，作成コストだけを考えるなら，虫を混じえないように注意するのにx円の費用がかかるとし，発生した虫を除去するのにy円の費用がかかるとした場合，$x>y$であるなら，ともかくプログラムを作成して，虫があれば取り除く方が安価であるといえよう．まして，虫取りソフトは，いずれにしても必要な道具であるから，その開発に注力することは決して無駄ではない．

筆者の個人的な意見では，従来ともすると「虫のないプログラム」が神聖化されたために，虫の存在を仮定する虫取りソフトは何か汚れたもの．本来存在すべきでないもの，として軽視されるという傾向が見られた．一般の平均的プログラマにとっては，虫の発生は，ある程度は許容されるべきで*，むしろ強力な虫取りソフトの導入とそれらを効果的に用いる訓練とが重要であると思う．対話的虫取りは後に述べるように強力なものであるから，この手法に習熟することがこれからは重要であろう．

特に強調されるべきは，プログラミング・システム内に虫取りソフトが組込まれ，系統的に使用でき，我我全体の共有財産化されてゆくことである．

3. プログラミング・システムについて

プログラミング・システムについては和田[32]，Sandewall[25]の解説があるので屋上屋を重ねるきらいがあるけれども，読者の便宜のためにその内容を極く簡単に述べようと思う．

◎ プログラミング・システムの定義

プログラムの開発を行うためのソフトウェアの統合的集合体，その構成要素としては，インタプリタ，コンパイラ，エディタ，虫取りソフト，解析プログラムなどがある．

◎ プログラミング・システムと TSS

時分割システム (TSS) は，本来オンラインで主計算機を共同で使用するためのものである．しかし，現在の TSS は，プログラムの開発を支援するためにコンパイラ，エディタや虫取りソフトなどを完備するようになっている．個々のソフト開発支援プログラムを統合すればプログラミング・システムとなるわけであるから，TSS は事実上プログラミング・システムに近いものにすることができる．その例としては Multics[4]，UNIX[24]，TENEX[2] などを挙げることができよう．

実際，Lisp プログラミング・システムの多くは，このような TSS 上で作成されている．

◎ プログラミング・システムと OS との関係

これは先ほどの TSS との関係とほぼ同じである．OS は，計算資源の有効活用を考慮するのだから，OS がプログラミング・システムの機能を取り込みないしは支援するというのは当然のことである．

また，支援形態としては，TSS（少なくとも対話的環境）が好ましいというのも当然のことである．プログラミング・システムを作成するには，OS 機能を利用できるようにすることが望ましい．

◎ プログラミング・システムと対話的環境の関係

システムの統合性という観点からだけでは，対話的環境の必要性は出てこないかもしれないが，各構成要素の活用，それも随時に種々の道具を活用してゆくという場面を考えると，どうしても対話的環境が必要となる．

すなわち，対話的環境を触媒にして，一見無関係ともみえる多くの機能（例えば，ファイル・システム，エディタ，コンパイラという組合わせなど）が，有機的に結合されうるのである．また，対話的環境の場合には，システムを統合化する際に余計な事柄を考えなくて済み，プログラミング・システムの建設が容易になるということも言える．

例えば，変数名の綴りの誤りをどうするのかというような点に関して，システムが暴走する危険性を犯さないで，ユーザの負担を大幅に軽減することができるのも対話的環境だからである．（第5章 DWIM の項目参照）

◎ プログラミング・システムとプログラミング言語の関係

プログラミング・システムを実現する場合には，その基本言語にはいくつかの要求事項が挙げられる[25]．重要なのは，プログラムをデータとして扱えること．

* To err is human という有名な言葉がある．

何らかのデータベースを持っていること，モジュール構造を有することなどである．これらの要件を満たすのは，Sandewall が述べているように，現在のところは Lisp, Snobol, Apl に限られるであろう．

しかしながら，「これ以外の言語はないのか」とか「これらの言語で充分なのか」という問いに関しては，将来，これらよりもっとプログラミング・システムに適した言語が開発される可能性のあることを述べておきたいと思う．実際，上述の言語は，いずれも「プログラミング・システム」などという概念が未だこの地上に存在しない頃に提案され，インプリメントされたものである．

4. Lisp プログラミング・システム

Lisp プログラミング・システムの成立理由とその特徴については他の文献 25), 32) に詳しいが，念のためにここで概略を述べることにしたい．

◎ 何故 Lisp が用いられたのか

① プログラムとデータの同一性——Modifiability

プログラムを処理するプログラムが容易に書ける．特に eval, apply が存在していることは重要である．

② プログラムを逐次的に実行することができる——Incrementality

Lisp は主としてインタプリタで稼動している* ので，各部分を次々と実行することによって最終目標に到達できるし，その間の過程がすべて明瞭にわかるようになっている．

③ プログラムが関数形で表現されている——Modularity

Lisp では基本手続きや制御文も含めて，すべての手続きが関数形で表現されているために，自然な形式でモジュール化が実現され，手続きの追加や変更が容易である．

④ データ構造に対する入出力が整備されている．

Lisp では文字列の読み込みだけでなく，リスト構造を読み込む入出力関数が用意されていて，プログラムの入出力にも利用することができる．

⑤ Lisp で書いたプログラムは，他の Lisp 上に移植することができる——Portability

Lisp は Fortran や Cobol のようには標準化されていないが，各方言間の変換が容易なので，Lisp のプログラムは，容易に他の Lisp 上に移すことができる．これは 1 人の作成したソフトウェア・ツールが，他の人々にとっても有用な共有財産となりうることを意味している．

◎ Lisp プログラミング・システムでの虫取り

① Lisp システムでは，対話的虫取りが行われている．

対話的 (interactive) 虫取りの機能については，鈴木[29]のまとめがある．

一般的に（3) 参照)，対話的虫取りはバッチ・システムの場合のような，非対話的な環境と比較して約 3 倍の効率のあることが指摘されている．筆者の個人的経験ではある程度の机上の虫取りと組み合わされた効果的な対話的虫取り手法はバッチ・システムの場合より少なくとも 10 倍能率が向上する．特に虫が難しいもの，再現困難なものである時ほど効果は大きい．

◎ Lisp システムは，他の言語やシステムにどのような影響をおよぼしているだろうか？

① 新しいプログラミング言語の出現

Lisp で書かれた Micro-planner[27] や Conniver[28] を別にしても，Lisp と同様，プログラミング・システムを目指した言語が出現している．その有名な例としては，Smalltalk[6] や Logo[23] を挙げることができる．この両者が，教育用のプログラミング言語であることは偶然とばかりは言えないであろう．プログラミングの全体像を理解するには，個々のコンパイラやインタプリタの個別的手法だけではなく，プログラムの変更，テスト，虫取りを含めたシステム的接近法が不可欠なのである．

② Lisp システムで良く使用される道具は他の言語にも用いられる．

端的な例としては，清書プログラム[9] を挙げることができる．これは Pascal[32], Fortran[11], Pl/1[35] で用いられている．他の例としては，虫取り用パッケージとか，エディタとのインタフェースなどを挙げることも出来る．

③ 最近のマイクロコンピュータの開発は，安価な対話的プログラミング・システムを可能にしつつある．

Lisp プログラミング・システムの有用性を認めて

* McCarthy, J.[10]は，Lisp インタプリタは所期の目標ではなく，偶然の所産だと述べている．

もLispが受けている制約，特に実行速度や記憶容量の制約からプログラミング・システムの実効性を疑問視する人がいる．最近のマイクロコンピュータを先端とする半導体技術は，Lispシステムを安価かつ高速に実現することを可能とするようになってきた．(cf. MITのLispマシンは約8万ドル，神戸大学で試験的に作成されたものは材料費だけだと250万円程度とか．) また，従来は大型機のTSSでしか享受できなかった対話的サポートもミニコンやマイクロコンの段階で安価に供給されるようになってきている．

5. 現在の各種Lispシステムにおける虫取りソフトの経験

この章では，Lisp 1.5[17]，Interlisp[31]，Lisp 1.9[12]，MIT Lisp Machine Lisp[33]の各Lispシステムに組込まれている虫取りソフトを紹介する．Lisp 1.5は，現在のLispの基となったものであり，InterlispはLispシステム中では最も強力な虫取りソフトを備えつけたシステムである．Lisp 1.9は筆者のかかわったLispシステムで，入出力の虫取りに工夫がこらされている．MIT Lisp Machineは本格的な専用パーソナルLispシステムとしては世界で初めてのシステムである*．

Lispシステムの開発は，標準的拘束案がないところから，各システム開発者の自由意志に任されており，多くの特色あるシステムが現在も作成されている．したがって，ここで紹介したシステムで，現存するLisp虫取りソフトをすべて網羅したとは言えないが，Lispユーザ界で一般的に良く知られているものは一応取り上げられている．将来，国内で多くの有意義な特色あるシステムが開発され，それらを紹介する機会を得られれば幸である．

5.1 Lisp 1.5

Lisp 1.5での虫取りソフトには，対話的システムという点を除いてはトレーサしかない．もっとも，今日でもトレーサしかないというLispシステムは多いし，トレーサを使えば大体虫は取れるというのも事実である．Lispでのトレーサの標準形は，

<関数名，実引数リスト，再帰レベル>——
関数に入った時
<関数名，　値　，再帰レベル>——
関数を出る時

となっている．このトレースを動的にしかも任意の時点にON, OFFできるのがLispシステムの特色である．他の言語で，ここまでトレースが取れるものは余り多くないので，羨ましがられることもある．また，トレース機能は虫取りだけでなくプログラムの解析にも役に立つ．このトレース機能を拡張してはどうかという話については後に第7章で述べる．

5.2 Interlisp

Interlispの虫取りソフトは，他に類をみない程充実している．したがって，例えばUCI-Lisp[31]のように，この虫取り機能を取り入れたもの (UCI-Lispの場合は，Stanford大学のLisp 1.6をベースにしている) が出てくるようになった．後述するLisp 1.9, Lisp Machine Lispでも1部の機能が取り入れられている．ただし，Interlispの虫取り機能のすべてが不可欠なものかどうかとか，対価格効率比の観点からみて良いかどうかには議論のあるところである．Interlisp自体は実験用プログラムの作成システムとしては非常に興味深いものである．その機能については Sandewall の議論[25]が有用であろう．

Interlispの虫取りシステムの基本機能はTeitelmanのPILOTシステム[30]に基づいている．

その基本機能には，次の3機能がある．

1) DWIN (Do-What-I-Mean)——自動綴り修正機能

計算機プログラムの虫で一番多いのがミス・スペルであろう．長いプログラムを流してみたら，1文字のミスでコンパイル・エラーだとか，実行エラーとか，無限ループとかに陥ったという経験は極めてありふれたものであろう．

DWIMは，ミス・スペルされた変数名，関数名，および括弧の文法ミスをシステム側が自動的に修正するというものである．この話だけを聞くと随分と胡散臭いと思う人もいるだろう．自動的にミス・スペルが修正できる筈がないという訳である．

Interlispマニュアルにのっている例を用いると図-2のようになる．

しかし，この仕掛けを聞いてみると，かなりまともな話で，人間工学的に見ても興味深い試みである．その方法は，次のようになっている．

まず，システムは，現在使用している変数名，ユーザ定義関数名，システムの組込変数名，組込関数名のリスト（正確には容量の定まった底抜けスタック）を常時管理している．ミス・スペルされた名前が入って

* Lispマシンそのものとしては，青山学院で開発されたAlps/1[36]の方が早く開発され，使用されている．

```
←DEFINEQ ((FACT (LAMBDA (N) (COND                    ←元の定義
((ZEROP N9 1) ((T (ITIMS N (FACCT 8SUB1 N]            (多くの誤り
(FACT)                                                 がある)
←
←PRETTYPRNT ((FACCT]
=PRETTYPRINT                                          }
=FACT                                                 } DWIM による修正
(FACT                                                 }
  [LAMBDA (N)
    (COND
      ((ZEROP N9 1)
        ((T (ITIMS N (FACCT 8SUB1 N])
(FACT)
←
←FACT (3]
N9 [IN FACT] -> N) ? YES                 ←括弧についてはユーザ
[IN FACT] (COND -- ((T --))) ->            の指示をあおいでいる
          (COND -- (T --))
ITIMS [IN FACT] -> ITIMES
FACCT [IN FACT] -> FACT
8SUB1 [IN FACT] -> (SUB1 ? YES
6
←PP FACT
(FACT                                                 ⎫
  [LAMBDA (N)                                         ⎪
    (COND                                             ⎪
      ((ZEROP N)                                      ⎬ 正しい定義になった
        1)                                            ⎪
      (T (ITIMES N (FACT (SUB1 N])                    ⎪
FACT                                                  ⎭
←
```

図-2 DWIM の実行例

くると，これは一般的には未使用変数もしくは未定義関数というエラーを生じる．(そうでない場合，すなわち,CAR と CDR を間違えた場合には，DWIM は無力である)

このエラーを生じた名前に最も近いものが，管理している名前表の中にあるかどうかを捜すのである．捜し方はスタックの上から順に調べてゆき，制限時間内に名前の違いの距離をあるアルゴリズムで計算するようになっている．この距離に上限があって，この範囲内に収まる候補者が制限時間内に唯一つ発見された場合には，これが，正しい名前だと思って計算を継続しようというのである．

このアルゴリズムには若干の知識が入っている．キーボート上でシフトの関係にある２文字，A と a，(と 8,) と９や，aa と a などの重ね打ちの文字は，距離が０であるとして計算される．これによって“8T”などという未定義関数を“(T”というストリングに戻す事により，文法ミスをも処理することが可能となるのである．

さらに，発見された名前はスタックのてっぺんに置き換えられるので，ユーザの良く間違えるパターンは比較的うまく処理されることになる．

この方式の特徴は，ミス・スペルをすべて修正しようとするのではなく，出来る範囲で処理しようというものである．さらに，ミス・スペルを修正した時にはその旨ユーザに報告する（端末に出力する）ので，ユーザはシステムが誤った解釈をした場合には，システムを停止させて暴走を食い止めることができる．(Interlisp は対話的システムなのでいつでも割込みをかけることができる)．

この DWIM 機能は，Interlisp を使用した時に最も印象に残る機能である．たとえば，デモンストレーション用プログラムを動かす時に，つい犯したタイプミスが DWIM 機能によりすんなりと受け入れられ実行されるのは，見ていても実行していても嬉しいものである．

2) ブレイク・パッケージ

エラーが生じた時，エラー処理専用のパッケージが起動されて，その中でエラーの診断や回復を行う事ができる．以前（例えば Lisp 1.6）の方法は，エラーに対してはユーザが，エラー状態受け取り関数 ERRSET を用いて，エラー後の分岐を前以って定めておくか，あるいはあっさりと top-level へ戻ってしまうしかなく，エラー状態を詳しく調べ，システムの状態を変更し，エラー発生時点から何らかの方法で処理を継続するということはできなかった．

このブレイク・パッケージもしくはエラー処理パッケージは，今日では多くの Lisp システムで採用されている．

このパッケージの名前がエラー処理パッケージとか，虫取りパッケージと名付けられていない理由の一つは，このパッケージは端末のブレイク・キーを押したり関数 BREAK を実行したりすることにより，ユーザが積極的に用いることができるからである．このような積極的使用法は，無限ループに入ったらしく思われるプログラムのチェックや，プログラムの暴走の防止，さらには複雑なプログラムの内部の理解といった面で役に立つものである．

3) ADVISE 機能

手続き（Lisp の場合にはすべて関数）の仕様を変更する場合には，通常その内容定義を変更せねばならない．対象となる関数が組込関数の場合には，アセンブラの変更が必要になるので，これは厄介な作業になる．

Interlisp の ADVISE 機能は関数の変更を容易に行

うために，関数の実行前後にある種の手続きを付加しようとするものである．すなわち，対象となる関数を $f(x)$ で表現すると，M_{before} と M_{after} という2種類の修飾関数を与えることにより f を f' に変更するのである．ここで $f'=M_{after}(f(M_{before}(x))$ となる．

この機能の利点は，アセンブラで書かれたシステム関数ですら容易に変更を加えることができるという点にある．

このアイデアを一般化すれば，黒川のクラス機能 (function class)[14] というような仕様定義システムを考えることができる．

以上の3機能の他に Interlisp では，次のような機能が実装されている．

4) Lisp 専用エディタ

このエディタはリストを直接に（文字列としてではなく）編集するもので Lisp 自身で書かれている．このエディタは関数定義やリストデータを直接変更できるので，ブレイク・パッケージでの使用に適している．

しかしながら，この Lisp エディタには議論もあって (Sandewall 参照)，むしろ普通のテキスト・エディタとファイルシステムを組合わせたソース・プログラム編集システムの方が良いのではないかという意見もある．

実際，この Interlisp エディタには，慣れないと使いにくい面もあるし，コメントの扱いやソース・リストの更新等の問題もあり，将来，解決されなければならない問題となっている．

5) 清書プリンタ[9]

現在では大抵の Lisp システムに装備されているので，特に目新しくないであろう．他の言語 (Fortran, Pl/1, Pascal, Pl 40 など) でも，清書プリンタを実装するのは常識化している．Lisp のソフトウェア工学全般への功績の一つと言うのは言い過ぎだろうか？

6) バック・トレーサ

Lisp システムでの経験から他の言語でももっと活用されて良いと思うものにバック・トレーサがある．これは，トップ・レベルからブレイクした（エラーを生じたり，ブレイク・キーを押されたりした）関数のレベルまでの呼び出しの系列（ネスティング）を出力するものである．このトレースは通常スタックのトップにある現在の関数から，それを呼び出した関数，さらにその関数を呼び出した関数というようにトップ・レベルまで逆向きにトレースするのでこの名前がある．

バック・トレーサがないと，ブレイクした関数が呼び出されるに至った経路を順番にトレースをかけて調べるということになる．バック・トレーサによって，どのようにして現在の状況に至ったかの概略が大体推測されるのである．

この機能は一部の Fortran コンパイラ（筆者の経験では TOSBAC 5600 のもの）にも取り入れられているが，再帰呼び出しを含むスタック方式の制御構造をもつ言語一般にとって強力なデバッグソフトとなる筈である．

7) 構造解析ルーチン

Interlisp には PRETTYSTRUCTURE というルーチンがあって*，関数間の呼び出し関係を出力する．Maclisp には関数間の呼び出しだけでなく，束縛変数，自由変数，GOTO タグなどの情報をも出力する INDEX パッケージがある．

これらの構造解析ルーチンは，実行前にプログラムに欠落や部分矛盾がないかをチェックするためだけでなく，Lisp コンパイラを走らせる前に無駄な部分がないかどうかとか，どの部分が最適化しうるかなどといった情報を得るために使用される．もちろん，プログラム全体への説明資料としての働きをも持つ．

このルーチンの特徴は，Lisp プログラムを対象とする場合には Lisp で容易にプログラム化できるという点にある．ユーザが作成したり，機能を強化したりした例が多い（Lisp 1.9 の場合には，電子技術総合研究所の元吉氏が作成したものが他の人にも利用されたりしている）．

8) UNDO 機能

Interlisp のコマンドには UNDO というものがあって（他に REDO もある），以前実行されたコマンドの内，指定されたものの取り消しができる．もちろんすべてのコマンドが取り消し可能というわけではないが，あるレベルでの関数定義や変数値やスイッチの変更，編集命令などのデータ構造の変更が取り消し可能である．

UNDO 機能を実現しているしかけは，これまでに入力したコマンドのリスト（実は底抜けスタック）をシステムが管理していて，取り消し可能なコマンドについては，そのコマンドの機能を打ち消すようなコマンドが貯えられている．UNDO は，この打ち消しコ

* 最新の Interlisp マニュアルからは消えている．(1974年，4月の第1版のマニュアルにはのっている)．

マンドを実行すれば良いのである．この機能はちょっとした誤操作や思い違いにより生じたエラーを取り消すのに便利である．対話的システムにおいては，このUNDO 機能を備えられれば便利なものが多い．例としてはエディタが挙げられよう．

5.3 Lisp 1.9

Lisp 1.9 は筆者が開発に携わったシステムであるが，他のシステムと比較してユニークな虫取り機能としては入出力用のものと，Lisp 1.9 システム自体の虫取り機能とが挙げられる．入出力機能の全貌に関しては 13) に詳しく述べられている．

1) 入出力デバッグ機能

ここに述べる機能は，Lisp 1.9 の「概念チャネル」(13) 参照) という新しい入出力単位に基づいている．

(i) ファイル入出力の端末における監視機能

ファイル入出力の最中にエラーが生じたような場合に役立つ．

(ii) 端末入出力のファイルへの保存機能

再現性に乏しい（と思われる種類の）エラーの実態を把握するために端末での会話の全セッションをファイルに保存しておくことは，非常に便利である．

入力部を別に保存しておけば，次の模擬テスト機能を用いて本当に再現性がないのかどうかもチェックできる．

(iii) ファイル中に貯えた入力系列により，端末入力の代行をさせる機能

これは定まったテスト系列を実施するのに有用である．

ユーザがこのテストをいちいち入力する場合には，思わぬタイプミスや誤操作等によってデバッグの手順を間違えたり，虫の行方を見失なうことがある．

(iv) トレース出力のファイル化

虫の所在がよくわからぬままトレースをかけると，膨大なトレース結果が出てウンザリすることが多い．場合によれば，肝心のエラー出現状態に至るまでにハードコピー用紙が切れたり，計算機の時間切れということで泣かされることもある．

ファイルに出力することにより，トレース実行の速度が早まり，後刻テキストエディタ等の力を借りて要領よく実状を把握することができる．

(v) 不必要な出力の停止

出力チャネルの中に「空出力チャネル」という特別のチャネルがあって，このチャネルへの出力は，実質上何の出力操作をも含まないようになっている．

この機能は，虫取りソフトとして本質的なものではないが，虫取りの過程を能率良くすすめる上で有用である．

2) 「TSS システム・デバッグ・トレース・パッケージ」との連結

他の多くの Lisp システムがそうであるように，Lisp 1.9 も完結したシステムというよりも「現在進行形」のシステムである．そのために Lisp 1.9 のようなシステムでは，出てきたシステムの虫を素早く処理することが絶対必要である．この目的のために Lisp 1.9 では TOSBAC-5600 の TSS 用の「システム・デバッグ・トレース・パッケージ」と連結するための関数 PTRCE (Physical TRaCEr の略) を組込んでいる．このパッケージは Lisp レベルというより，機械語レベルでの虫取りソフトである．したがって，スナップショット，パッチ，ブレイクポイント，レジスタ参照等といった機能をもっている．さらに良いことには，各種のトレース機能があり，レジスタやメモリの参照，変更を監視できるし，統計機能があってシステムの性能測定に役立てることもできる．

5.4 Lisp Machine Lisp[33]

ここで紹介する Lisp Machine は MIT のものである．他にも国内および国外で数多くの Lisp Machine プロジェクトがあり，性能的または価格的に MIT の Lisp Machine より優れたものもある．

しかし，MIT の Lisp Machine の特長は，システムの基本機能であるオペレーティング・システムやユーティリティである画面制御やエディタまで，すべて Lisp を用いて作成しているところにある．すなわち，いわゆる高級言語機械の一つの終点を示しているとも言える．

MIT Lisp Machine は，MIT Maclisp の影響を色濃く残しているが，虫取り機能として重要なのは次の2点であろう．

1) エラー処理用関数の開放

Lisp Machine でのエラー処理は，ユーザが細部に至るまで規定することが可能である．多くの handler が備えつけられているとともにユーザが使い易いように開放されている．

エラーが出てからの虫取りというより，エラーの出現と同時にいろいろな処理をしてしまえるようになっ

ている.

2) MAR 機能

これは Lisp 1.9 で見られたシステム・レベルのデバッガを Lisp レベルに持ってきたものと言える.

すなわち, Lisp Machine のメモリ管理機能に直接指示できるのである. したがって, メモリの参照変更について Lisp の言葉 (NIL, T その他) を直接用いることが出来ると同時に, この機能を働かせてもそれ程大きな処理速度の低下は見られない. MAR 機能はマイクロ命令のトラップ機能により実現されている.

6. 虫取りソフトのまとめ

前章で紹介した各種の虫取りソフトを分類してまとめてみよう.

大きく分類すると虫をみつけるためのもの, つぶすためのもの, 自動虫取りを試みるものの三つに分けられる.

6.1 虫を見つけるためのもの

この中に入るのは Lisp 1.5 のトレーサ, Interlisp のブレイク・パッケージ, 清書プリンタ, バック・トレーサ, 構造解析ルーチン, Lisp 1.9 の入出力虫取り機能, TSSシステム・デバッグ・トレース・パッケージのもつトレースやスナップショット機能, Lisp Machine のエラー処理関数や MAR 機能といったものである.

虫取りの第一段階がこの虫を見つけることであるし, この段階の用具には種類も経験も多い.

6.2 虫を除去するための用具

Interlisp のアドバイス機能, エディタ, UNDO, Lisp 1.9 のシステム・デバッガのパッチ機能, そして Lisp Machine Lisp のエラー処理用関数などがこれに当たるであろう.

虫を除去するための処理は, 最初に述べたように, プログラムを変更するための処理に他ならない. この変更処理はアドバイス機能やエラー処理用関数を見てもわかるように, 同時にプログラミング・システムや言語の機能を拡張するために用いることができる. また, この変更の容易さはシステムのモジュール性が前提になる.

6.3 自動的に虫を見つけて除去するもの

現在のところシステムとしてまとまっているのは Interlisp の DWIM だけである. Lisp Machine Lisp のエラー処理ルーチンは, ユーザがこの種のシステムを組むための道具として利用しうるものである.

将来, 虫のない自動プログラミング・システムがどうなるかは, 今のところ不明であるが, ユーザの与えた仕様をチェックして, 虫を自動的に除去したり, 出来たプログラムをチェックしてはその中の虫を除くというような, この種の虫取りソフトが必要となるのではないかと思われる.

特に人間——機械複合体による自動プログラミングでは, 人間側の不注意による虫の発生が常に伴うためにこの種の虫取りソフトが必要になる.

ただし, DWIM よりもっと高度なもの, すなわち, 完全な自動虫取りシステムが可能かどうかは判断が難しい. 結局, 自動プログラミングと自動デバッギングはほぼ同程度の困難さをもつであろうということは, 相互の処理の関連などから言えるのだが, どちらも決して容易な問題ではない.

自動虫取りの研究に関しては 7), 28) などがある. さらには, Pretty Reader[10] のように, プログラム入力時の文法エラーを未然に防ぐようになったものをも, これに含めることができるかもしれない.

7. 将来のプログラミング・システムにおける虫取りソフト

将来のことを話すのは, 決して難しいことではない. しかしながら, 将来について述べたことが実現されるのは決して並の事柄ではない. ここでは, 筆者自身の感じている要求と, ここ 2, 3 年来の動向をベースにして予想を立ててみる. これらが当るかどうかは残念ながら保証の限りではないが, これらの多くは時間と労力が割ければ, 是非実現したいと思っているものである.

1) 要求仕様段階での虫取りソフト

要求仕様定義の重要性は近年認識されてきた. 要求仕様から正しいプログラムを作成する技術が確立したとしても, 要求仕様段階での虫取りが必要となる.

2) 自動化虫取りソフト

虫を見つけてはプログラムを修正して, 虫を除いてしまう自動化された虫取りソフトが DWIM 以外にも出現する. DWIM より高機能なものが出来ると良い.

3) より進んだエディタ

これが完全だというようなエディタはまだ存在していない. Interlisp 式のエディタと画面用のエディタを組み合わせたものが, 必要であることはわかっている[25]のだが, 具体的にどのようなものが良いのかはまだわかっていない. いろいろな試みとその評価が必要

である.

4) 強力なトレーサ

既に述べたようにトレーサは Lisp では古典的な虫取りソフトであるが，現在のものより進んだより強力なトレーサが欲しい．例えば，条件付きのトレース，条件によるブレークポイント，条件付きのスナップショット，そして，一般的に有用な統計情報収集などが出来れば大いに役に立つ.

スピード・ダウンを招かないためには，Lisp machine 的な設計方式が必要である．このような強力トレーサは，ある程度知的なプログラム・モニタと考えることもできる.

5) 知的なプログラム解析ルーチン

現在，既にプログラム解析ルーチンが作成されているが，人間にとってより見易く，コンパイラその他のプログラムにとってより有用な情報を与えるような解析ルーチンの開発が必要である.

6) 環境設定ルーチン

一般的には，一つのエラーの背後には複数の虫が存在している．この虫をすべてつぶしてゆくには，いろいろな状況を徐々に変更してゆく必要がある．そのためには，現在の環境を保存し，しばらく実行した後に再びこの環境を回復することができる必要がある.

これは AI 用言語 (Conniver 19) etc.) では，context mechanism と呼ばれているものである．現在，一部のシステムではファイルへのダンプとロードという形式で実現されているが，context-tree のごとき形式で扱えることが望ましい.

7) 虫を未然に防ぐための道具

Pretty Reader[10] のように，文法エラーを出せないようなシステムがあれば役に立つ．プログラム作成時のエディタ中に DWIM 機能があって，ミス・スペルを修正してくれるのも良い．行き着く先は自動プログラミングだが，それ以前に道具として使用できるのも多いのではなかろうか.

8. 他のプログラミング言語では

第1章のはじめに述べたように，本稿の主対象は Lisp ユーザ以外の人々である．それらの人々の使用しているプログラミング言語（システム）に対して本稿の果しえた（いと思っている）目標をまとめてみよう.

1. 虫取りソフトは，プログラミング・システムの一部として有機的に結合され，埋め込まれねばならない.

   （ただし，前提条件として，使用言語のプログラミング・システム化が必要である）

2. 虫取り用のシステムは対話的であるべきである．対話的環境下での虫取りソフトは面倒な保護処理を省略されることにより，単純かつ強力なものとなりうる.

   （ただし，前提条件としてプログラミング・システム自体が対話的であることを要する）

3. Lisp の虫取りソフトですら完全なものではなく，まだ改良の余地がある．したがって，各言語での道具の開発で独創性を発揮する余地は充分にある．ただし，改良の方向としては，自動虫取り的な虫の発見と虫の駆除とを組み合わせたものが主流となろう.

4. 虫取りソフトの重要部分としては，エディタとのリンク（ないしはエディタの取り込み）を考えるべきである.

5. プログラム解析機能が一般的に必要となる．また，この解析結果を他の部分（例えばコンパイラなど）で利用することが望ましい.

6. DWIM 的機能を取り入れてみてはどうか？

   ミス・スペルが自動修正されれば有難いと思うのは，インタプリタの場合よりもコンパイラの方である．1回のコンパイル時間を丸々損するかどうかを考えてみればよい.

   もちろん危険性もあるのだが，Interlisp を見ている限りでは，むしろ利点の方が多いと思える.

   ただし，これも最初の項目に関係するのだが，Lisp のように，内部データベースを持ってないとこのような仕掛けすらできない．ともかく，DWIM 自体は Lisp に固定されるべき理由は何らない筈である.

以上述べられた項目を実現するために，それでは何をすれば良いかということは，今の時点では筆者には答えられない事柄である．個々の具体化作業はむしろ読者諸氏（嬢）に任せられるべきものであろう．ただし，一つの示唆として，Lisp プログラミング・システムの使用経験をもってみることを推奨することができる.

9. おわりに

この解説中では，最初に虫取りとプログラミング・システムについて述べた．虫取りは残念ながら将来に

も不可欠な処理過程である．したがって，虫取りソフトは将来にわたって拡充し，機能強化する必要がある．

Lisp の虫取りソフトで強調したいことは，虫取りソフトをそれ単独で設計するべきではなく，プログラミング・システム全体の一部分として位置づけるべきだということである．これは，プログラミング・システムの他の部分（場面）からも虫取りソフトを使用できるようにすべきであるということと，虫取り用にプログラミング・システムの他の部分を利用できるようにすべきであるという二つの事を意味している．

Lisp の虫取りソフトは将来のプログラミング・システムにどのような虫取りソフトが含まれるべきかについて示唆を与えてくれるものである．

この解説が Lisp 以外の言語によるプログラミング・システムを作成しようとしている人々に役立てば幸いである．

参考文献

1) Allen, J. R.: Anatomy of LISP, McGraw Hill. (1978).
2) Bobrow, D. G. et al.: TENEX, a Paged Time Sharing System for the PDP-10 CACM, 16, 3 (Mar. 1972).
3) Brooks, F. P. Jr.: The Mythical Man-Month Addison-Wesley (1975).
4) Corbato, F. J. and Vyssotsky, V. A.: Introduction and Overview of the Multics System Proc. FJCC (1965).
5) Dijkstra, E. W.: A Discipline of Programming Prentice-Hall (1976).
6) Goldberg, A. and Kay, A. (eds): Smalltalk-72 Introduction Manual Xerox PARC, SSL 76-6 (1976).
7) Goldstein, I. P.: Understanding Simple Picture Programs MIT AI-TR 294 (Sep. 1974).
8) Guttarg, J. V.: The Specification and Application to Programming Abstract Data Types Univ. of Toronto, CSRG-59 (1975).
9) 長谷川 洋：「LISP Pretty Printer」，記号処理研究委員会資料 (1976)．
10) 長谷川 洋：「LISP Pretty Reader について」，情報処理学会，第17回全国大会予稿集 (1976)．
11) Kernighan, B. W. and Plauger, P. J.: Software Tools Addison-Wesley (1975).
12) 黒川利明：LISP 1.9 プログラミング・システム，情報処理 17, 11 (Nov. 1976).
13) Kurokawa, T.: Input/Output Facilities in LISP 1.9 SOFTWARE-Prac. & Exp., 8 (1978).
14) Kurokawa, T.: An Informal Introduction to Function-class: A Programmable Specification Technique Unpublished Memo (Jan. 1979).
15) Kurokawa, T.: Lisp Activities in Japan to Appear in Proc. 6th IJCAI at Tokyo (1979).
16) 離数譜「Lisp 手習」，bit Vol. 10-11 (1978-1979)．
17) McCarthy, J. et al.: LISP 1.5 Programmer's Manual MIT Press (1966).
18) McCarthy, J.: History of Lisp ACM SIGPLAN Notices, 13, 8 (Aug. 1978).
19) McDermott, D. V. and Sussman, G. J.: The Conniver Reference Manual MIT AI Memo 259a (1974).
20) Moon, D. A.: MACLISP Reference Manual MIT Project MAC (Aug. 1974).
21) 中西正和：「LISP 入門—システムとプログラミング」，近代科学社 (1977)．
22) Naur, P. et al.: Software Engineering Mason/Charter Pub. Inc. (1976).
23) Papert, S. A.: Teaching Children Thinking Programmed Learning and Educational Technology, 9, 5 (1972). (残念ながらこの文献は現在持っていない)
24) Ritchie, D. M. and Thompson, K.: The UNIX Time Sharing System CACM, 17, 7 (1973).
25) Sandewall, E.: Programming in an Interactive Environments: the "LISP" Experiences ACM Comp. Surveys, 10, 1 (1978).
26) Siklóssy, L.: Let's Talk LISP Prentice-Hall (1976).
27) Sussman, G. J. et al.: Micro-planner Reference Manual MIT AI Memo 203A (1973).
28) Sussman, G. J.: A Computational Model of Skill Acquisition MIT AI-TR 297 (Aug. (1973).
29) 鈴木則久：「インタアクティブ・デバッガの効用」，bit Vol. 11, No. 3 (1979).
30) Teitelman, W.: Toward a Programming Laboratory Proc. 1st IJCAI (1969) also in [22].
31) Teitelman, W.: Interlisp Reference Manual Xerox PARC (1974).
32) 和田英一：「ソフトウェア開発ツールの最近の傾向」，ソフトウェア工学シンポジウム報告集(1979年1月)．
33) Weinreb, D. and Moon, D.: Lisp Machine Manual 2nd Preliminary Version MIT (Jan. 1979).
34) TOSBAC-5600 TSS System Debug and Tracer, TOSHIBA.
35) indent command in: Multics Programmers Manual, commands and active functions HIS AG92 (Jan. 1975).
36) 井田昌之他：「ALPS からの2, 3の話題」，情報処理学会記号処理研究会資料 (1979年3月)．
37) Bobrow, R. J., Burton, R. R. and Doryle, L.: UCI LISP MANUAL.

(昭和54年7月5日受付)