計算機実習を効果的に行なうための モニターシステムの試み[1]

津田塾大学 数学科
鈴木悦子, 小川貴英

1/12/1993

計算機実習を効果的に行なうには, 順調に課題をこなしている, あるいはトラブルを起こして困っているといった実習中の学生の状況を, 実時間で把握する必要がある. 我々はワークステーション上で Emacs を使って, プログラムの編集, コンパイル, 実行を繰り返して実習を行なっている学生の, エラー情報, キー入力量などに注目し, 学生がどこで困っているのか, 問題のある学生は誰かといったことをできるだけ正確に把握し, それを実習にフィードバックする, モニターシステムを試作した. 今回は, このモニターシステムのしくみとモニターシステムを使用して収集した学生情報の解析について報告する.

1 はじめに

計算機実習を効果的に行なうには, 学生の状態把握は不可欠である. 一方, 計算機実習は数十人規模でも教師に, 助手1～2名で行なわれることが多い. この様な状況では, 実習中の学生すべてに個別に対応することはできないし, 目が届く範囲は限られてくるので, 学生全体の状況把握も容易でない. この問題を解決するために, 実習中の学生の状況を表示するモニターシステムを試作した.

限られた時間内で実習を効果的に行なうためには, 次にあげるような情報が必要になる.

1. 全体の進捗状況
   プログラムを入力中, デバッグ中といった学生の状況を, 全体的に把握をすることで, 学生の理解度, 問題の難易度が分かる. 難し過ぎたときにはヒントを与える, 易し過ぎて早く作業を終ってしまいそうなときには, 別の問題を与えるなどの対応が可能になる.

2. 行き詰まっている学生の状況
   多くの学生が同じような誤解をして, 類似の失敗を繰り返しているといった状況は, 早い時期に検出できれば, 適切な助言を与えることが可能になる.

3. 特に問題のある学生
   教師が個別に対応できる学生の数は限られるので, 行き詰まってどう

[1] A monitor system for effective programming workshop, Etsuko Suzuki, Takahide Ogawa, Department of Mathematics, Tsuda College

しようも無くなっている学生など, 問題がありそうな学生を検出できればそれらの学生は個別に指導できる.

これらの情報を実時間で得るために, 学生の実習状況をモニターする必要がある. モニターする情報は, 多い方がより正確に学生の状況を把握できる可能性があるが, 実時間処理をするにはあまり多くの情報があっても処理し切れない. 学生の状況把握に役に立つ情報としては, 次のようなものが考えられる.

1. プログラムの編集状況
   ファイル名, ファイルサイズ, 内容, キー入力, 作業時間.

2. コンパイル状況
   コンパイラを起動した時刻, コンパイルの結果, エラーメッセージ.

3. 実行状況
   実行した時刻, 実行の結果.

これらのうちで, プログラムの内容, 実行結果は意味の解析までやらないと役に立たないので, 採っても有効に使うのは難しい. キー入力の状況は, 学生の作業状況が分かるので有効であるが, 1 文字単位で見ると情報が多過ぎる. 適当な間隔で入力数とキーの種類を見ることにすれば処理可能である.

以上のような考察を元に, 実習状況のモニターシステムを構築した. 以下,2 節でモニターシステムの概要を説明し,3 節でモニターシステムで得られたログの解析結果を述べる.

2 モニターシステム

ここで考えるモニターシステムは, プログラム言語の入門教育で行なう実習を対象とし, ネットワークで接続されているワークステーションを想定している.

モニターシステムは, 次の 3 つのプログラムからなる.

1. 学生の情報をサーバーに送るクライアント

2. 学生情報を収集しログファイルに落とすサーバー

3. サーバーが収集した情報を教師の画面に表示する表示プロセス

今回はプロトタイプを作るだけなので, 言語は Pascal に限定し, 編集には Emacs を使い, コンパイルと実行も Emacs の中で行なう. 学生は段付けなどを自動的に行なうように設定した pascal-mode[2] で編集する. 学生情報を採る部分はこの pascal-mode のプログラムに手を入れて, 情報を送ることにした. 今回のシステムでは, 変更の容易さなどを考慮して, 情報の転送にはメールを利用した. 学生情報を収集するプログラムなど, 上記以外の部分は,perl で記述してある. 処理速度は速いとはいえないが,30 名程度のクラスで実験した限りでは, 特に問題は起きていない.

以下の情報を学生ごとに採る.

1. キーストローク数 (1 分毎)

2. コンパイル (終了時)

3. 実行 (開始時)

キーストローク数は,Emacs Lisp が保持しているキー入力情報を利用して計算する. Emacs Lisp は常に最後のキー入力を 100 文字を保持している. これを利用すれば,1 分間に 100 文字までのキー入力は調べられる. 1 分間に 100 文字以上入力されると,100 文字に切り

[2] ©Inge Frick, inge@nada.kth.se

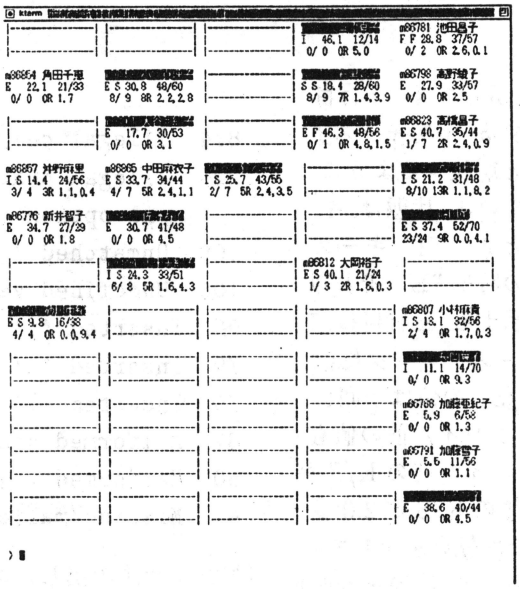

図 1: モニター画面

捨てられてしまうが, 学生のタイプの速度が遅いので 100 文字を越える場合はほとんどない. 実際, 学生のタイプ速度を調べてみたところ, 平均 83.6 ストローク / 分, 標準偏差 25.2 であった. 実習中の入力は, この値より更に落ちるので, 100 文字で十分であると思われる. コンパイル終了時には, コンパイラ起動時刻, エラーメッセージを送り, 実行開始時には, 実行の開始時刻を送る. ファイル名, ファイルサイズは全ての場合に付随する. ファイル名, ファイルサイズは, Emacs のバッファ名, バッファサイズを用いる.

モニター画面は 2 つあり,1 つは学生全体を表示する画面 (図 1), もう 1 つは, エラーメッセージを多い順に表示する画面である. 二つの画面はコマンドで切り替えられる. 学生全体の表示は, ワークステーションの配置順に表示される. 以下に表示内容を示す. "で囲まれたものは, そのまま表示される文字である.

1. ユーザー ID(学籍番号)
2. 名前
3. 状態
   ("S"|"F"|"R"|"E"|"I")
   "S":compile success,
   "F":compile fail,
   "R":run,"E":edit,"I":idle
4. 前回のコンパイル結果
   ("S"|"F"|" ")
   " ": コンパイルをしていない
5. 平均キーストローク数
6. キーストローク数が 0 のキーログ数 "/" キーログの総数
7. コンパイル成功回数 "/" コンパイル回数
8. 実行回数 "R"

9. K/S (キーストローク総数 / ファイルサイズ)

1行目は順に,1,2 を表示する. 2行目は3～6までを表示する. 平均キーストローク数は今の状態を反映するように, 傾きつき平均をとっている. 状態は, キーストローク数が0のときに"I", キーストローク数が0以外の時は"E", それ以外は最新の情報を表示する. 三行目は, 7～9を表示する. キーストローク総数 / ファイルサイズ (K/S) の値は一回目のコンパイル後は, コンパイル前の値も表示をしている. コンパイル前の K/S が 3.0 以上, コンパイル後の K/S が 2.0 以上になった人は 1,2 を反転表示をしている.(K/S の意味については, 次節を参照)

3 ログファイル解析

1,2 年生の実習中にサーバーが書き込んだログファイルを解析をした.1 年生は 5 月と 6 月に採った記録,2 年生は 10 月と 11 月に採った記録を使った.1 年生の記録は本システムのプロトタイプを使って採ったので, キーストロークの情報はない.

3.1 エラーメッセージの解析

1 回の実習で 1 人あたりのコンパイル回数は表 1のようになる.

	回	コンパイル回数	構文エラー	
			なし	あり
1 年	1	18.9	7.6	11.3
	2	13.7	7.9	5.8
2 年	1	11.2	8.8	2.4
	2	12.4	8.6	3.8

表 1: コンパイル回数の比較

1 年生と 2 年生を比較すると, コンパイル回数は 1 年生が上まわっている. 1 年生の回数が多いのは構文エラーが多

```
814  Illegal character
220  Illegal character, use the
     -xl option to process % XXX
108  Unmatched ' for string
106  Undefined variable
95   Inserted ';'
79   Inserted ','
79   Inserted ')'
62   Malformed declaration
60   Malformed statement
42   Missing/malformed expression
```

図 2: 1 年生の出したエラーメッセージ (1 回目)

```
27  Undefined variable
24  expression has invalid type
21  Inserted identifier
20  Malformed var declaration
17  Left operand of + must be
    integer, real or set, XXXX
16  Write requires an argument
15  Type clash: real is
    incompatible with integer
15  Inserted '['
14  Undefined identifier
14  Inserted ']'
```

図 3: 1 年生の出したエラーメッセージ (2 回目)

いためであり,構文エラーなしの回数は1年,2年の間で大きな差異は認められない.1回の課題で構文エラーなしのコンパイル回数は,課題によってもあまり変わらないことがわかる.

1年生の構文エラーは課題に依存して大きく変わる. 1年生の1回目の課題は入出力と簡単な計算,2回目は配列を使ったプログラムであった.図2,3はその時のエラーメッセージを多い順に並べたものである.1回目では,文字列に関するエラーメッセージが多くなり,2回目では,配列に関するエラーメッセージが多くみられる.文字列のエラーメッセージは漢字コードがかかわっていたので意味がわからずエラー回数が増えている.

一般に,1年生は文法をきちんと理解していないため,エラーメッセージの意味を正しく読みとれないので,正しい対応ができず,同じ誤りを繰り返す傾向がみられる.

2年生の場合,課題によってコンパイル回数は変わらない. ある程度構文を理解しているので,課題にかかわらず,コンパイルエラーは,すぐに修正できる.

エラーメッセージを見るだけでも1年生の状態はほぼ把握できるが,2年生になると構文を理解しているので,エラーメッセージ解析だけでは,状態を把握するには不十分であることがわかる.

3.2 キー入力の解析

2年生はエラーメッセージ解析だけでは,十分に状況を把握できないため,他の情報も必要となる.我々はキー入力の情報を使うことにした.

課題を与えられた学生の典型的な行動は以下の3つの部分からなる.

1. プログラムを考える
2. プログラムを入力する
3. デバッグする

1ではキー入力は発生しない. 2と3では,キー入力の仕方は,大きく変わる.

ファイル入力とデバッグの境界をどこにするかという問題があるが,ここでは,1回目のコンパイルの前までをファイル入力時,それ以降をデバッグ時とした.

実際に,学生のログを見ると,ファイル入力時は,文字キーが中心に入り,コントロールキーは文字キーに対して少ない.デバッグ時は,削除,移動に使われているコントロールキーが多く入っており,境界は妥当であると考えられる.

単純に学生のキーストローク数を追跡すると,学生のキー入力の有無と頻度はわかるが,学生のタイプ速度に値が依存するので,キー入力の多少が学生の活動度を正確に反映していない.キーストロークの総数を見ていると,学生の作業量は見えてくるが,キーストローク総数は課題のプログラムの大きさによって異るので,動きは,課題ごとに値が変動してしまう.

以上のことを考慮し,学生のタイプ速度,ファイルサイズの影響を消すために,キーストローク総数をファイルサイズで割ったもの(この値を以降K/Sと表記する)を使って学生の有効作業量を比較した. この値は,キーストロークがどれくらい有効にファイルサイズに反映されているかを示す値になっている.入力した文字が全てバッファに入ってくれば,この値は1になる.この値が1以下になるのは,自動段付け,yankでコピーしたときなどでのみ起こる. キーストローク総数はデバッグ開始時に0に戻して計算してある.K/Sを縦軸にして,横軸に時間をとった場合の1学生の入力開始から終了までを図4,5に示す. このグラフはキー入力がそのままファイルサイズに反映されるときは,1に近づく.

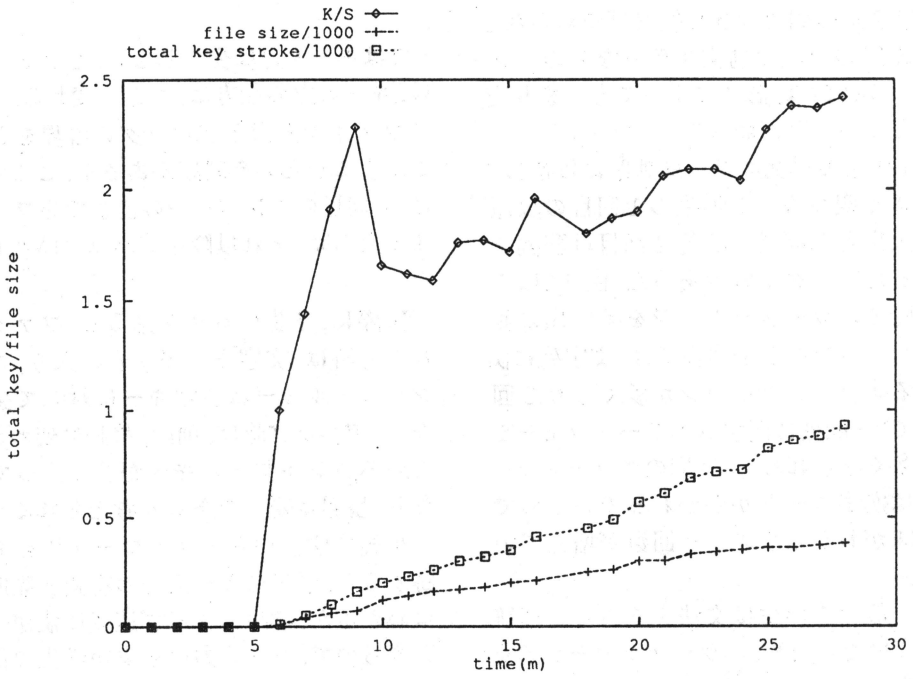

図 4: ファイルサイズと総キーストローク数(プログラム入力中)

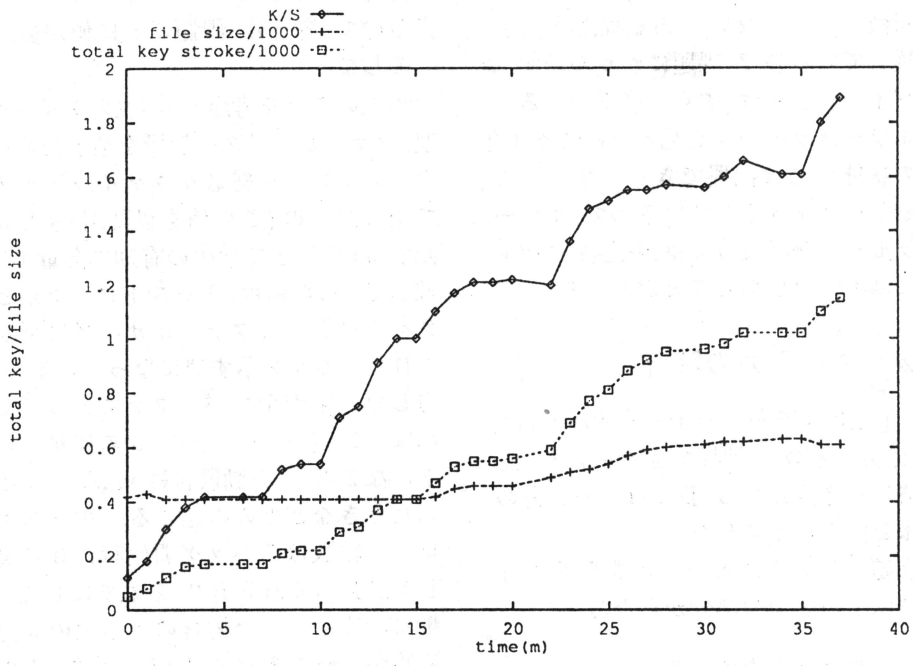

図 5: ファイルサイズと総キーストローク数(デバッグ中)

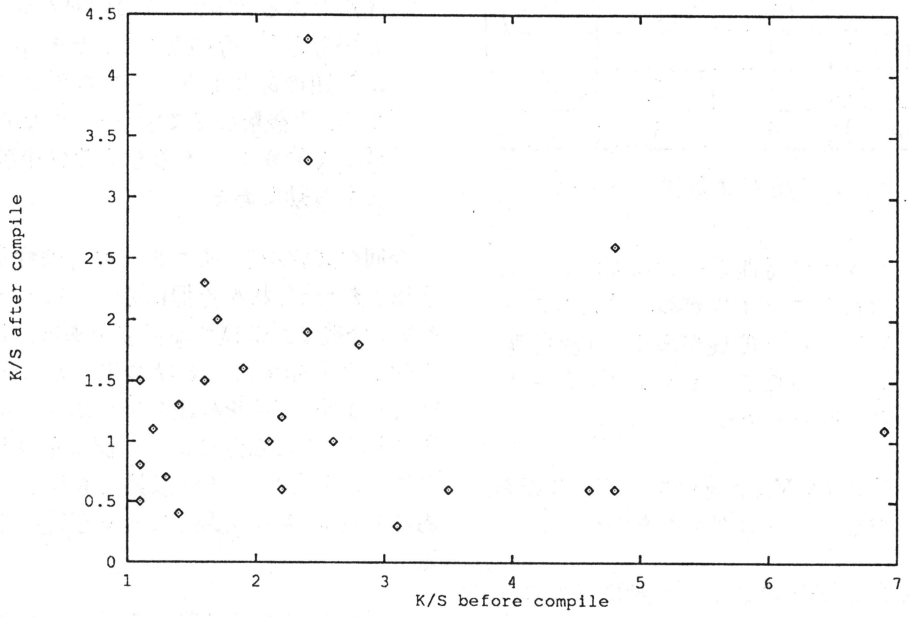

図 6: プログラム入力時とデバッグ時の K/S 値の関係

何もしないときは横這いになる. カーソル移動の制御文字が多い場合には, キー入力だけが増えるので増加する. 通常デバッグ中は, 単調増加する.

2 年生の 2 回目の実習のときのログを使って K/S 値を求めてみた. プログラム入力終了時とデバッグ終了時の K/S 値の人数分布表を示す.(表 2,3)

プログラム入力終了時を縦軸, デバッグ終了時を横軸にしてプロットしたものが図 6である. 表 4は, これをそれぞれプログラム入力終了時を A,B,C, デバッグ終了時を X,Y,Z にわけ 3 × 3 のグループに分けたものである.

表 4の各グループには次のような特徴が見られる.

1. A × (X,Y): 入力時間が短く無駄がない. デバッグも短い.
2. B × (X,Y): ある程度, 論理的な方針がたち, 考えながら入力している.
3. (A,B) × Z: 入力時間は短いが, デッ

K/S	人数
0 ～	0
1 ～	11
2 ～	8
3 ～	2
4 ～	3
5 ～	0
6 ～	1

表 2: ファイル入力終了時の分布

K/S	人数
0 ～	9
1 ～	11
2 ～	3
3 ～	1
4 ～	1

表 3: デバッグ終了時の分布

	X(0 ～)	Y(1 ～)	Z(2 ～)
A(1 ～)	4	5	2
B(2 ～)	1	5	2
C(3 ～)	4	1	1

表 4: K/S によるグループ化

バッグに時間をとられている．入力したファイルが論理的にわかっていない可能性がある．(教科書，前に作ったファイルなどを流用した可能性もある)

4. C × (X,Y,Z): 書いては消しながら，少しずつプログラムを作る．

A × (X,Y) の学生は一般に優秀であるが，他人のプログラムを写した人もここ (特に A × X) に分類される．両者は実時間でキー入力の様子を見れば区別できる．

(A,B) × Z,C × (X,Y,Z) といったグループに入る学生には，少し注意が必要であると考えられる．

ここで切り分けている数値は，課題によって変化することが予想される．しかし，ファイル入力時に 3.0 を越える，または，デバッグ時に 2.0 を越える学生は，少し問題があるといえる．

4 おわりに

今回試作したモニターシステムを使った実験の結果，次のようなことが分かった．

1. 学生全体の進捗状況は，コンパイルの状況とキーストロークの状況で，かなり把握できる．

2. 初心者の状況はエラーメッセージからもかなり分かる．同じ種類のメッセージが多発するときは，特に注意がいる．

3. 問題のある学生は，ほぼ把握することができる．今のところ，できるのは問題のありそうな学生の集合になる．事後解析まですると，かなり詳しく分かる．その意味で事後解析も有効である．

今回の実験から，モニターの有効性が確認されたが，状況把握は完全とはいえない．今後まだ改良する必要がある．K/S で学生を分類することはエディタの使い方の上手，下手を検出しているように思われる．これ以外のよい指標は今のところわからない．それ以外にも以下にあげる点は，今後実験してみる予定である．

1. 対象に応じて収集するデータを変更する
   対象とする学生に応じて，収集する情報は変更できると都合がよい．

2. 処理速度の改善
   今後，他のデータを採ろうとすれば，処理速度の改善が必要である．それには，専用サーバーを作る必要がある．