CHAPTER 8

Software Tools

Computing involves two main activities—program development and use of application software. Language processors and operating systems play an obvious role in these activities. A less obvious but vital role is played by programs that help in developing and using other programs. These programs, called *software tools*, perform various housekeeping tasks involved in program development and application usage.

Definition 8.1 *A* software tool *is a system program which*

1. *interfaces a program with the entity generating its input data, or*
2. *interfaces the results of a program with the entity consuming them.*

The entity generating the data or consuming the results may be a program or a user. Figure 8.1 shows a schematic of a software tool.

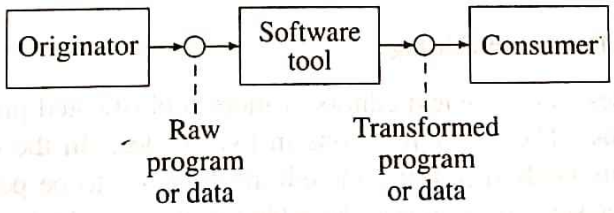

Fig. 8.1 A software tool

Example 8.1 A file rewriting utility organizes the data in a file in a format suitable for processing by a program. The utility may perform blocking/deblocking of data (see Section 16.5.2), padding and truncation of fields and records, sorting of records, etc. The file rewriting utility is a software tool according to Part 1 of Definition 8.1.

In this chapter we discuss two kinds of software tools—software tools for program development, and user interfaces.

8.1 SOFTWARE TOOLS FOR PROGRAM DEVELOPMENT

The fundamental steps in program development are:

1. Program design, coding and documentation
2. Preparation of programs in machine readable form
3. Program translation, linking and loading
4. Program testing and debugging
5. Performance tuning
6. Reformatting the data and/or results of a program to suit other programs.

Step 3 requires use of language processors. All other steps involve transformations of programs or data which fall within the purview of Definition 8.1. The software tools performing these transformations are described in the following Sections.

8.1.1 Program Design and Coding

Two categories of tools used in program design and coding are

1. Program generators
2. Programming environments.

As described in Section 1.2.1, a *program generator* generates a program which performs a set of functions described in its specification. Use of a program generator saves substantial design effort since a programmer merely specifies *what* functions a program should perform rather than *how* the functions should be implemented. Coding effort is saved since the program is generated rather than coded by hand. A *programming environment* (see Section 8.4) supports program coding by incorporating awareness of the programming language syntax and semantics in the language editor.

8.1.2 Program Entry and Editing

These tools are text editors or more sophisticated programs with text editors as front ends. The editor functions in two modes. In the *command mode*, it accepts user commands specifying the editing function to be performed. In the *data mode*, the user keys in the text to be added to the file. Failure to recognize the current mode of the editor can lead to mix up of commands and data. This can be avoided in two ways. In one approach, a quick exit is provided from the data mode, e.g. by pressing the `escape` key, such that the editor enters the command mode. The `vi` editor of Unix uses this approach. Another popular approach is to use the screen mode (also called the what-you-see-is-what-you-get mode), wherein the editor is in the data mode most of the time. The user is provided special keys to move the cursor on the screen. A stroke of any other key is taken to imply input of the corresponding character at the current cursor position. Certain keys pressed along with the `control`

key signify commands like erase character, delete line, etc. Thus end of data need not be explicitly indicated by the user. Most Turbo editors on PC's use this approach.

Editors are discussed in greater detail in Section 8.2.

8.1.3 Program Testing and Debugging

Important steps in program testing and debugging are selection of test data for the program, analysis of test results to detect errors (if any), and debugging, i.e. localization and removal of errors. Software tools to assist the programmer in these steps come in the following forms:

1. *Test data generators* help the user in selecting test data for his program. Their use helps in ensuring that a program is thoroughly tested.
2. *Automated test drivers* help in *regression testing*, wherein a program's correctness is verified by subjecting it to a standard set of tests after every modification. Regression testing is performed as follows: Many sets of test data are prepared for a program. These are given as inputs to the test driver (see Fig. 8.2). The driver selects one set of test data at a time and organizes execution of the program on the data.

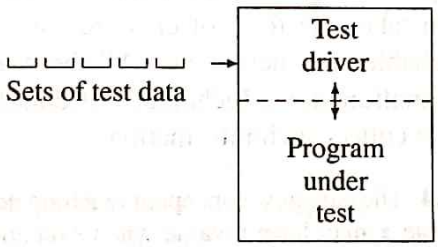

Fig. 8.2 Automated test driver

3. *Debug monitors* help in obtaining information for localization of errors.
4. *Source code control systems* help to keep track of modifications in the source code.

Test data selection uses the notion of an *execution path* which is a sequence of program statements visited during an execution. For testing, a program can be viewed as a set of execution paths. A test data generator determines the conditions which must be satisfied by the program's inputs for control to flow along a specific execution path. A test data is a set of input values which satisfy these conditions.

Example 8.2 The execution path involving the statement `a := a + 1.0;` in Fig. 8.3 is traversed during execution only if $x > z$. To test this path, the test data generator must select a value for y such that $x > z$, e.g. $y = z^2$.

Two problems exist in automated testing of programs. One concerns wastage of testing effort on infeasible paths, i.e. paths which are never followed during the

```
x := sqrt(y) + 2.5;
if x > z then
   a := a + 1.0;
else
   ...
```

Fig. 8.3 Program for Example 8.2

execution of a program. The second concerns testing of complex loop structures. It is best to leave both these aspects to manual testing. Thus automated testing can be used to show up a first batch of errors at low cost. This would free the programmer and system analyst to tackle more complex aspects of testing.

Example 8.3 In Fig. 8.3, if a statement `z := y + 10;` precedes `x := sqrt(y) + 2.5;` then the statement `a := a + 1.0;` lies along an infeasible path.

Producing debug information

Classically, localization and removal of errors has been aided by special purpose debug information. Such information can be produced statically by analysing the source program or dynamically during program execution. Statically produced debug information takes the form of cross reference listings, lists of undefined variables and unreachable statements, etc. All these are useful in determining the cause of a program malfunction. Techniques of data flow analysis (see Section 6.5.3.2) are employed to collect such information.

Example 8.4 The data flow concept of reaching definitions can be used to determine whether variable x may have a value when execution reaches statement 10 in the following program

```
No.      Statement
                ...
10.      sum := x + 10;
                ...
```

If no definitions of x reach statement 10, then x is surely undefined in statement 10. If some definition(s) reach statement 10, then x *may* have a value when control reaches the statement. Whether x is defined in a specific execution of the program would depend on how control flows during the execution.

Dynamically produced debug information takes the form of value dumps and execution traces produced during the execution of a program. This information helps to determine the execution path followed during an execution and the sequence of values assumed by a variable. Most programming languages provide facilities to produce dynamic debug information.

Example 8.5 Figure 8.4 illustrates the trace and dump facility supported by most Fortran compilers. `debug unit` specifies the kind of dynamic debug information desired.

and the file in which it is to be written. The `at` specification is used to enable or disable the production of `trace` and `display` data when control reaches a specified statement during program execution.

1. `debug unit` (<*file_number*>, <*list_of_options*>)

   Debug output is written in the file <*file_number*>.
   <*list_of_options*> can contain:

`trace`	:	Trace of labelled statements executed
`subtrace`	:	Trace of subprograms called
`init` (<*list*>)	:	Trace of assignments made to each variable in <*list*>
`subchk` (<*list*>)	:	Subscript range check: Report error if subscript in any reference to an array in <*list*> is out of bounds

2. `at` <*label*>

   Indicated actions are executed when statement bearing <*label*> is encountered during execution

`trace on`	:	Trace is enabled
`trace off`	:	Trace is disabled
`display` <*list*>	:	Values of variables in the list are written in the debug file

Fig. 8.4 Trace and dump facilities in Fortran

Use of trace and dump facilities is cumbersome due to the volume of information produced. To improve the effectiveness of debugging, it should be possible for the user to dynamically control the production of debug information, i.e. to dynamically specify the trace or dump actions. These facilities can be provided by the language compiler or interpreter, or by a language independent tool.

Example 8.6 Figure 8.5 illustrates the use of dynamic debugging facilities supported by many Basic interpreters. The programmer can set breakpoints at many statements using the `stop on` command. The system initiates a debug conversation with the programmer when control reaches any breakpoint during program execution. During a debug conversation, the programmer can display or change values of variables and set or remove breakpoints.

A debug monitor is a software which provides debugging support for a program. The debug monitor executes the program being debugged under its own control. This provides execution efficiency during debugging. It also enables the monitor to perform dynamically specified debugging actions. A debug monitor can be made language independent, in which case it can handle programs written in many languages. The dynamic debugging technique (DDT) of DEC-10 is a well known example of this approach. The working principles of debug monitors are discussed in Section 8.3.

1. Breakpoint and Dump facilities:
   (a) `stop on` <*list of labels*>
   Sets breakpoint(s) at labelled statement(s).
   (b) `dump at` <*label*> <*list of variables*>
   Dumps values of variables at given label
2. Interactive debugging facilities :

   These commands can be issued when program reaches a breakpoint

`display` <*list of variables*>	:	Displays values of variables
`set` <*var*> = <*exp*>	:	Assigns value of <*exp*> to <*var*>
`resume`	:	Resumes execution
`run`	:	Restarts execution

Fig. 8.5 Debugging facilities in Basic

8.1.4 Enhancement of Program Performance

Program efficiency depends on two factors—the efficiency of the algorithm and the efficiency of its coding. An optimizing compiler can improve efficiency of the code but it cannot improve efficiency of the algorithm. Only a program designer can improve efficiency of an algorithm by rewriting it. However, this is a time consuming process hence some help should be provided to improve its cost-effectiveness. For example, it is better to focus on only those sections of a program which consume a considerable amount of execution time. A performance tuning tool helps in identifying such parts. It is empirically observed that less than three percent of program code generally accounts for more than 50 percent of program execution time. This observation promises major economies of effort in improving a program.

Example 8.7 A program consists of three modules A, B and C. Sizes of the modules and execution times consumed by them in a typical execution as given in Table 8.1.

Table 8.1

Name	*# of statements*	*% of total execution time*
A	150	4.00
B	80	6.00
C	35	90.00

It is seen that module C, which is roughly 13 percent of the total program size, consumes 90 percent of the program execution time. Hence optimization of module C would result in optimization for 90 percent of program execution time at only 13 percent of the cost.

A *profile monitor* is a software tool that collects information regarding the execution behaviour of a program, e.g. the amount of execution time consumed by its modules, and presents it in the form of an *execution profile*. Using this information, the programmer can focus attention on the program sections consuming a significant amount of execution time. These sections can be improved either through code optimization (see Chapter 6), or through improvement of the algorithm.

Example 8.8 Figure 8.6 shows a sample execution profile. The column '# of executions' in the profile indicates the number of times a statement was executed. '# of true' indicates how many times the condition in an `if` statement was found to be true. '% time' gives the percentage of execution time spent in executing a statement, the body of a loop, or a module.

```
Program alpha execution profile

program name    % execution time
main program        11.15
    sub1            41.59
    sub2            15.54
    meor            31.72

Program 'meor' consumed 31.72% of total time

                    statement                    # of executions   % time   # of true
      subroutine meor(eoper)                          5           31.72
      integer eoper(10)                               0
      k = 9                                           5            0.08
      do 10 i1 = 1,10                                39           30.74
      j = i1 + k                                     39            4.35
      if(sw.eq.1) goto 50                            39            9.53        3
      if(eoper(j).eq.bl) goto 15                     36            4.44       26
      if(eoper(j).eq.zero) goto 20                   10            2.06        4
      pr = eoper(j)                                   6            0.21
      goto 10                                         6            0.21
15    ...
      ...
10    continue                                       39            2.01
      ...
      end                                             5            0.59
```

Fig. 8.6 An execution profile

The profile indicates that subroutine `meor` consumed 31.72 percent of the total execution time. The DO loop in `meor` consumed 30.74 percent of the time consumed by subroutine `meor`. This shows that it is important to optimize the loop. It is further seen

that the three `if` statements consume more than half of the loop execution time. It is possible to interchange the second and third `if` statements, however this is counter-productive since the second `if` is mostly true while the third `if` is mostly false (see '# of true' column). Hence there appears to be little scope for optimization of `meor` short of replacing the algorithm by a better one. An analysis of the profile would reveal this information in a couple of minutes, thereby saving considerable time and effort.

Information regarding execution counts, etc. is obtained by introducing counters in the program or in its generated code. This can be done by the language compiler as in IITFORT [Dhamdhere *et al*, 1979] or by a preprocessor [Ingalls, 1972].

8.1.5 Program Documentation

Most programming projects suffer from lack of up-to-date documentation. Automatic documentation tools are motivated by the desire to overcome this deficiency. These tools work on the source program to produce different forms of documentation, e.g. flow charts, IO specifications showing files and their records, etc.

8.1.6 Design of Software Tools

Program preprocessing and instrumentation

Program preprocessing techniques are used to support static analysis of programs. Tools generating cross reference listings and lists of unreferenced symbols; test data generators, and documentation aids use this technique. *Program instrumentation* implies insertion of statements in a program. The instrumented program is translated using a standard translator. During execution, the inserted statements perform a set of desired functions. Profile and debug monitors typically use this technique. In a profile monitor, an inserted statement updates a counter indicating the number of times a statement is executed, whereas in debug monitors an inserted statement indicates that execution has reached a specific point in the source program.

Example 8.9 A debug monitor instruments a program to insert statements of the form

```
call debug_mon (const_i);
```

before every statement in the program, where `const_i` is an integer constant indicating the serial number of the statement in the program. During execution of the instrumented program, the debug monitor receives control after every statement. Assume that the user has specified the following debug actions:

```
at 10, display total
at 20, display term
```

Every time debug monitor receives control, it checks to see if statement 10 or 20 is about to be executed. It then performs the debug action indicated by the user.

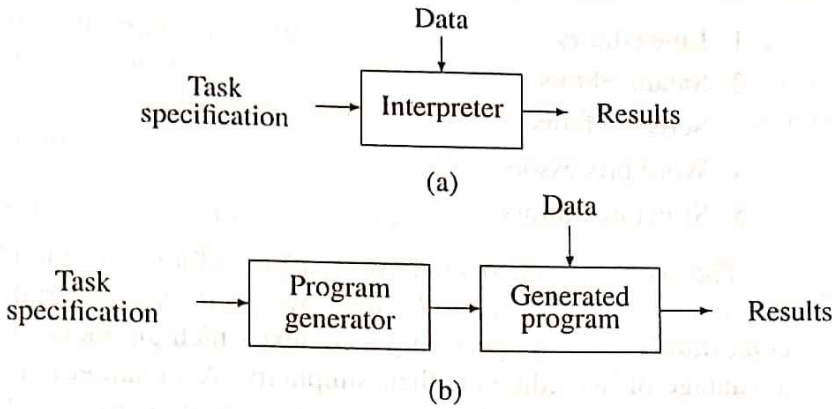

Fig. 8.7 Software tools using interpretation and program generation

Program interpretation and program generation

Figure 8.7 shows the schematic of software tools using the techniques of interpretation and program generation. Use of interpreters in software tools is motivated by the same reasons that motivate the use of interpreters in program development, viz. absence of a translation phase in processing a program. Since most requirements met by software tools are ad hoc, it is useful to eliminate the translation phase. However, interpreter based tools suffer from poor efficiency and poor portability, since an interpreter based tool is only as portable as the interpreter it uses. A generated program is more efficient and can be made portable.

Example 8.10 An organization specializing in the development of interactive applications needs to design user friendly screens for form fillin functions (see Ex. 1.2). It has been decided to use a software tool to obviate repeated coding efforts.

Use of the interpretive approach involves development of a general purpose interpreter for screen handling (we will call it the *screen handler*). Every application that needs to use the tool will have to interface with the screen handler. Whenever the application needs to use a screen, it will have to call the screen handler with the specification of the screen. If the program generator schematic is used, specification of the screen will have to be input to the program generator. The generated program will be in the form of a function which will become a part of the application.

From Fig. 8.7 it can be seen that the interpreter, besides being slow in execution, will require more memory. It will, however, permit dynamic changes in the specifications. The program generator provides execution efficiency. It also provides code space efficiency because only a routine tailored to a desired screen format needs to exist in the application. Any change in the specification of a screen will now imply generation of a new function and its integration with the application.

8.2 EDITORS

Text editors come in the following forms:

1. Line editors
2. Steam editors
3. Screen editors
4. Word processors
5. Structure editors

The scope of edit operations in a line editor is limited to a line of text. The line is designated *positionally*, e.g. by specifying its serial number in the text, or *contextually*, e.g. by specifying a context which uniquely identifies it. The primary advantage of line editors is their simplicity. A stream editor views the entire text as a stream of characters. This permits edit operations to cross line boundaries. Stream editors typically support character, line and context oriented commands based on the current editing context indicated by the position of a *text pointer*. The pointer can be manipulated using positioning or search commands.

Line and steam editors typically maintain multiple representations of text. One representation (the *display form*) shows the text as a sequence of lines. The editor maintains an *internal form* which is used to perform the edit operations. This form contains end-of-line characters and other edit characters. The editor ensures that these representations are compatible at every moment.

8.2.1 Screen editors

A line or stream editor does not display the text in the manner it would appear if printed. A screen editor uses the what-you-see-is-what-you-get principle in editor design. The editor displays a screenful of text at a time. The user can move the cursor over the screen, position it at the point where he desires to perform some editing and proceed with the editing directly. Thus it is possible to see the effect of an edit operation on the screen. This is very useful while formatting the text to produce printed documents.

8.2.2 Word Processors

Word processors are basically document editors with additional features to produce well formatted hard copy output. Essential features of word processors are commands for moving sections of text from one place to another, merging of text, and searching and replacement of words. Many word processors support a spell-check option. With the advent of personal computers, word processors have seen widespread use amongst authors, office personnel and computer professionals. Wordstar is a popular editor of this class.

8.2.3 Structure Editors

A structure editor incorporates an awareness of the structure of a document. This is useful in browsing through a document, e.g. if a programmer wishes to edit a specific

function in a program file. The structure is specified by the user while creating or modifying the document. Editing requirements are specified using the structure. A special class of structure editors, called syntax directed editors, are used in programming environments.

Example 8.11 NLS (short form for 'on line system') is an early structure editor [Engelbart, English 1968] oriented towards document editing. A document has a hierarchic structure, with different levels like group, plex, branch and statement. Items within a higher level item are siblings (i.e. brothers) with predecessor-successor relationships. Thus, one can talk of a group, the preceding or succeeding group, first statement in a group, etc.

Contemporary editors support a combination of line, string and screen editing functions. This makes it hard to classify them into the categories defined in this section. The `vi` editor of Unix and the editors in desk top publishing systems are typical examples of these.

8.2.4 Design of an Editor

The fundamental functions in editing are travelling, editing, viewing and display. Travelling implies movement of the editing context to a new position within the text. This may be done explicitly by the user (e.g. the line number command of a line editor) or may be implied in a user command (e.g. the search command of a stream editor). Viewing implies formatting the text in a manner desired by the user. This is an abstract view, independent of the physical characteristics of an IO device. The display component maps this view into the physical characteristics of the display device being used. This determines where a particular view may appear on the user's screen. The separation of viewing and display functions gives rise to interesting possibilities like multiple windows on the same screen, concurrent edit operations using the same display terminal, etc. A simple text editor may choose to combine the viewing and display functions.

Figure 8.8 illustrates the schematic of a simple editor. For a given position of the editing context, the editing and viewing filters operate on the internal form of text to prepare the forms suitable for editing and viewing. These forms are put in the editing and viewing buffers respectively. The viewing-and-display manager makes provision for appropriate display of this text. When the cursor position changes, the filters operate on a new portion of text to update the contents of the buffers. As editing is performed, the editing filter reflects the changes into the internal form and updates the contents of the viewing buffer.

Apart from the fundamental editing functions, most editors support an undo function to nullify one or more of the previous edit operations performed by the user. The undo function can be implemented by storing a stack of previous views or by devising an inverse for each edit operation. Multilevel undo commands pose obvious difficulties in implementing overlapping edits.

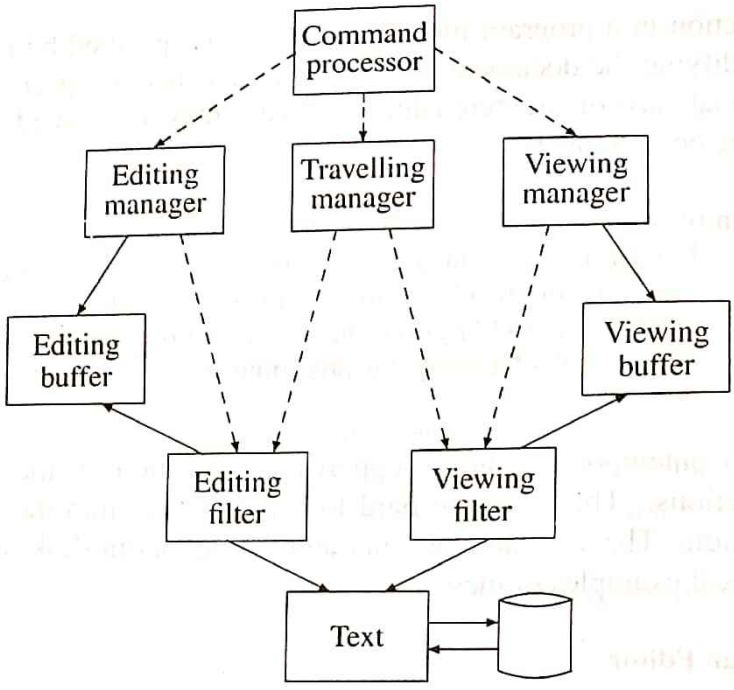

Fig. 8.8 Editor structure

8.3 DEBUG MONITORS

Debug monitors provide the following facilities for dynamic debugging:

1. Setting breakpoints in the program
2. Initiating a debug conversation when control reaches a breakpoint
3. Displaying values of variables
4. Assigning new values to variables
5. Testing user defined assertions and predicates involving program variables.

The debug monitor functions can be easily implemented in an interpreter. However interpretation incurs considerable execution time penalties. Debug monitors therefore rely on instrumentation of a compiled program to implement its functions. To enable the use of a debug monitor, the user must compile the program under the debug option. The compiler now inserts the following instructions:

A few `no-op` instructions of the form
`no-op` <*statement no*>

before each statement, where <*statement no*> is a constant indicating the serial number of the statement in the program. The compiler also generates a table containing the pairs (variable name, address). When a user gives a command to set a breakpoint at, say, statement 100, the debug monitor instruments the program to introduce the instruction

<*SI_instrn*><*Code*>

in place of the `no-op` instructions preceding `no-op 100`.

The compiled code for the program executes directly on the CPU until it reaches an <*SI_instrn*>. Execution of the <*SI_instrn*> produces a software interrupt with the interrupt code <*Code*>. This signifies a debug interrupt. (Alternatively the debug monitor can use a `BC ANY, DEBUG_MON` instruction instead of <*SI_instrn*>). The debug monitor now gains control and opens a debug conversation. The user may ask for display or modification of program variables. This is implemented using the (variable name, address) information produced by the compiler.

The sequence of steps involved in dynamic debugging of a program is as follows:

1. The user compiles the program under the `debug` option. The compiler produces two files—the compiled code file and the debug information file.
2. The user activates the debug monitor and indicates the name of the program to be debugged. The debug monitor opens the compiled code and debug information files for the program.
3. The user specifies his debug requirements—a list of breakpoints and actions to be performed at breakpoints. The debug monitor instruments the program, and builds a *debug table* containing the pairs (statement number, debug action).
4. The instrumented program gets control and executes up to a breakpoint.
5. A software interrupt is generated when the <*SI_instrn*> is executed. Control is given to the debug monitor which consults the debug table and performs the debug actions specified for the breakpoint. A debug conversation is now opened during which the user may issue some debug commands (which are implemented through interpretation) or modify breakpoints and debug actions associated with breakpoints. Control now returns to the instrumented program.
6. Steps 4 and 5 are repeated until the end of the debug session.

Unix supports two debuggers—`sdb` which is a PL level debugger and `adb` which is an assembly language debugger. `debug` of IBM PC is an object code level debugger.

8.3.1 Testing Assertions

A debug assertion is a relation between the values of program variables. An assertion can be associated with a program statement. The debug monitor verifies the assertion when execution reaches that statement. Program execution continues if the assertion is fulfilled, else a debug conversation is opened. The user can now perform actions to locate the cause of the program malfunction. Use of debug assertions eliminates the need to produce voluminous information for debugging purposes.

8.4 PROGRAMMING ENVIRONMENTS

A programming environment is a software system that provides integrated facilities for program creation, editing, execution, testing and debugging. It consists of the following components:

1. A *syntax directed editor* (which is a structure editor)
2. A language processor—a compiler, interpreter, or both
3. A debug monitor
4. A dialog monitor.

All components are accessed through the dialog monitor. The syntax directed editor incorporates a front end for the programming language. As a user keys in his program, the editor performs syntax analysis and converts it into an intermediate representation (IR), typically an abstract syntax tree. The compiler (or interpreter) and the debug monitor share the IR. If a compiler is used, it is activated after the editor has converted a statement to IR. The compiler works incrementally to generate code for the statement. Thus, program execution or interpretation can be supported immediately after the last statement has been input. At any time during execution the programmer can interrupt program execution and enter the debug mode or return to the editor. In the latter case he can modify the program and resume or restart its execution.

The main simplification for the user is the easy accessibility of all functions through the dialog monitor. The system may also provide other program development and testing functions. For example, it may permit a programmer to execute a partially completed program. The programmer can be alerted if an undeclared variable or an incomplete statement is encountered during execution. The programmer can insert necessary declarations or statements and resume execution. This permits major interfaces in the program to be tested prior to the development of a module. Some programming environments also support reversible execution, whereby a program's execution can be 'stepped back' by one or more statements.

The Cornell Program Synthesizer

The Cornell Program Synthesizer (CPS) [Teitelman, Reps, 1981] is a syntax directed programming environment for a subset of PL/I known as PL/CS. It contains a syntax directed editor, a compilation-and-interpretation schematic for program execution, and a collection of debugging tools.

The editor guides the user to develop a well structured program through the use of templates. A *template* is a unit of information concerning PL/CS. It consists of some fixed syntax information and a few placeholders. A *placeholder* signifies a position in the program where a specific language construct, e.g. a statement or an expression, may be inserted by the programmer. (It is equivalent to a nonterminal symbol in a sentential form.) A program development session starts when the user keys in the command `main`. The editor now displays the template for a program, viz.

```
/* comment */
name : procedure options(main);
  {declaration}
  { statement }
  END name;
```

where ☐ marks the position of the cursor, and { declaration } and { statement } are placeholders. The user can bring the cursor to a placeholder and begin to expand it by indicating the kind of entity he wishes to put there. CPS now replaces the placeholder by the template for that entity. Use of templates avoids syntax errors. Thus, errors like more **else**'s than **if**'s or missing **end**'s for **do** statements cannot arise in a CPS generated program.

Example 8.12 A user wishes to place an **if** statement in the placeholder { statement } in the initial template. When he indicates this, the program becomes

```
/* comment */
name : procedure options(main);
   {declaration}
   if {condition} then {statement}
                  else {statement}
   END name;
```

The user can move the cursor to the new placeholder {statement}, indicate that an assignment statement is to be inserted in its place and type `a = b;` followed by the `return` key. The placeholder { statement } is now replaced by the statement `a = b;`. Moving the cursor to { statement } in the **else** part and pressing `return` indicates that the **else** clause is empty.

As the user keys in program statements, they are converted into an IR. Cursor position on the screen is mapped into the correct node of IR to support expansion of a placeholder. CPS supports execution of a partially complete program. When an unexpanded placeholder is encountered during execution, the user is prompted to expand it before resuming execution. The screen is split into two during execution. The lower half is used to display execution results and debug output, if any. The upper half displays a part of the user program in which the statement under execution is marked with the cursor symbol ☐. Thus, as execution proceeds, the flow of execution through the program is displayed on the screen. The user can interrupt execution at any moment and open a debug conversation or modify the program before resuming or restarting execution.

Experience with the use of CPS has uncovered some significant strengths and weaknesses. The template governed approach guarantees that every CPS generated program is syntactically valid. However, it makes certain operations, like insertion of a loop in an existing program, quite difficult. Another significant drawback is the absence of an undo facility. Later programming environments have removed some of these deficiencies. For example, the programming environment Poe offers unlimited

undo facilities. It also offers on line help. The user can query the various expansion options available at a placeholder and the scope and accessibility of variables at a given point in the program. All these facilities further enhance the utility of programming environments.

EXERCISE 8.4

1. Comment on the statement "Dynamic debugging is easier to implement in interpreters than in compilers".
2. Comment on whether you would prefer a generative schematic or an interpretive schematic for following purposes:
   (a) Implementing `display` commands issued during dynamic debugging
   (b) Producing a report from a file
   (c) Writing a general purpose screen handling system
   (d) Handling data base queries.

   Give reasons for your answers.
3. A program profiling package is to be implemented for a Pascal compiler. The following options are being considered for its design:
   (a) Write a preprocessor which would introduce Pascal statements in a program to collect information during execution of the program,
   (b) Support a `profile` option in the compiler to perform program instrumentation.

   Which option would you use? Why?

8.5 USER INTERFACES

A *user interface* (UI) plays a vital role in simplifying the interaction of a user with an application. Classically, UI functionalities have two important aspects—issuing of commands and exchange of data. In early days of computing, a user was often the application designer or developer. Hence an understanding of commands and data was implicit in the use of an application. In those days UI's did not have an independent identity. This situation changed because of two reasons. First, as applications became larger a user was no longer expected to know all details concerning an application. Hence presentation of commands and prompts for data became important. Second, as applications grew to newer fields, it became necessary to assume a lower level of computer skills in an application user. This increased the importance of UI's by adding a new aspect, viz. on line help, to their functionality. The on line help component of the UI serves the function of educating the user in the capabilities of an application and the modalities of its usage. A well designed UI can make the difference between a grudging set of users and an excited user population (and even a fan following!) for an application.

A UI can be visualized to consist of two components—a *dialog manager* and a *presentation manager*. The dialog manager manages the conversation between the user and the application. This involves prompting the user for a command and

transmitting the command to the application. The presentation manager displays the data produced by the application in an appropriate manner on the user's display or printer device.

8.5.1 Command Dialogs

Commands are issued to an application through a command dialog Three ways to implement command dialogs are:

1. Command languages
2. Command menus
3. Direct manipulation.

Command languages for computer applications are similar to command languages for operating systems. Primitive command languages support imperative commands with the syntax <*action*> <*parameters*>. More sophisticated command languages have both declarative and imperative commands and a syntax and semantics of their own. A practical difficulty in the use of a command language is the need to learn it before using the application. This implies a large commitment of time and effort on the part of the user, which makes casual use of the application forbidding. On line help can provide some relief by avoiding the need to memorize the syntax of commands, however it cannot eliminate the need to invest time and effort in initial learning of the command language.

Command menus provide obvious advantages to the casual user of an application, as the basic functionalities of the application are reflected in the menu choices. A hierarchy of menus can be used to guide the user into the details concerning a functionality. Interesting variations like pull-down menus are designed to simplify the use of menu systems. Most Turbo compilers use pull down command menus.

A *direct manipulation system* provides the user with a visual display of the universe of the application. The display shows the important objects in the universe. Actions or operations over objects are indicated using some kind of pointing device, e.g. a cursor or a mouse.

Example 8.13 `Lotus 1-2-3` is an example of a direct manipulation system. The universe of the application is the spreadsheet. The user can select a specific cell in the spreadsheet, change its contents and study the effect of this change on other cells.

Examples of direct manipulation can also be found in screen editors and video games.

Principles of command dialog design

Psychologists and human factors engineers have formulated a set of principles to ensure the effectiveness of command dialogs. Some of these are:

1. Ease of use

2. Consistency in command structure
3. Immediate feedback on user commands
4. Error handling
5. On line help to avoid memorizing command details
6. Undo facility
7. Shortcuts for experienced users.

Command menus score over command languages on the basis of principles 1, 2 and 5, while command languages are superior on the basis of principle 7. Some menu systems provide a *type ahead* facility while entering menu choices. This permits the user of a hierarchical menu system to type the menu choices for menus which are yet to be displayed on the user screen. This improves the appeal of the menu system for experienced users. Recent trends in command dialog design have been towards the use of direct manipulation systems with a graphics capability.

8.5.2 Presentation of Data

Data for an application can be input through free form typing. Alternatively, a form fillin approach may be used (see Ex. 1.2) when large volume of data is involved. Application results can be presented in the form of tables. Summary data can be presented in the form of graphs, pie charts, etc.

8.5.3 On Line Help

On line help is very important to promote and sustain interest in the use of an application. It minimizes the effort of initial learning of commands and avoids the distraction of having to consult a printed document to resolve every doubt. On line help to the users can be organized in the form of on line explanations, demonstrations, tutorials or on line manuals concerning commands. The on line help facility should be organized such that desired information can be efficiently located using the structure of the information, e.g. by searching for section headings, subtitles, figure or table names, footnotes, etc. Another effective method of organizing on line help is to provide context sensitive help, whereby a query can fetch different responses depending on the current position of the user in the application.

Hypertext

Hypertext visualizes a document to consist of a hierarchical arrangement of information units, and provides a variety of means to locate the required information. This takes the form of

1. Tables and indexes
2. String searching functions
3. Means to navigate within the document
4. Backtracking facilities.

Effectiveness of a hypertext document depends on the care with which it is organized. Hypertext authoring systems have been developed to help the application designer in the design of hypertext documents.

8.5.4 Structure of a User Interface

Figure 8.9 shows a UI schematic using a standard graphics package. The UI consists of two main components, presentation manager and dialog manager. The *presentation manager* is responsible for managing the user's screen and for accepting data and presenting results. The *dialog manager* is responsible for interpreting user commands and implementing them by invoking different modules of the application code. The dialog manager is also responsible for error messages and on line help functions, and for organizing changes in the visual context of the user.

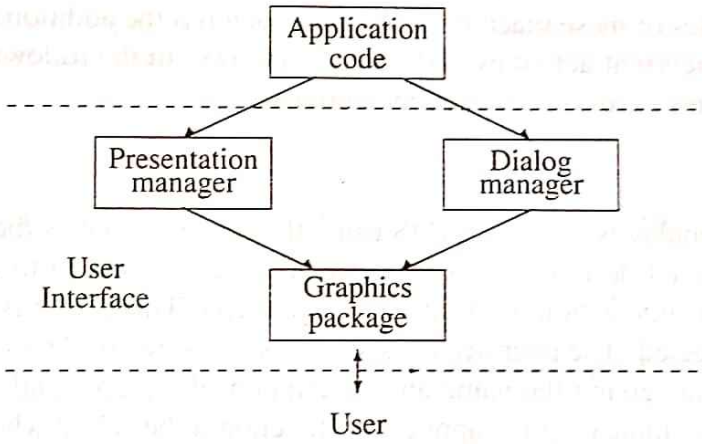

Fig. 8.9 User Interface

Example 8.14 The fields Employee name and Married in the form fillin screen of Fig. 1.8 (see Ex. 1.2) can be displayed by issuing the following commands to the graphics package:

```
Display ((2,5) Employee name :)
Drawbox ((2,25)(55,1))
Display ((10,5) Married :)
Drawdoublebox ((10,25)(2,1)(yes))
```

The dialog monitor is trivial in this application.

8.5.5 User Interface Management Systems

A *user interface management system* (UIMS) automates the generation of user interfaces. The UIMS accepts specification of the presentation and dialog semantics to produce the presentation and dialog managers of the UI respectively. The presentation and data managers could be generated programs specific to the presentation

and dialog semantics. Alternatively, the presentation and dialog managers could use interpretive schematics to implement the presentation and dialog semantics.

A variety of formalisms have been used to describe dialog semantics. These include grammars, event descriptions and finite state machines. In a grammar based description, the syntax and semantics of commands are specified in a YACC like manner. Thus, the interface is activated when a user types in a command. The event based approach uses a visual model of the interface. A screen with icons is displayed to the user. Selection of an icon by clicking the mouse on it causes an event. The action specified against the event is now performed.

The grammar and event description approaches lack the notion of a sequence of actions. The finite state machine approach can efficiently incorporate this notion. The basic principle in this approach is to associate a finite state machine with each window or each icon. Actions are specified on the basis of conditions involving the states of these machines. This approach has the additional power of coordinating the concurrent activities in different windows. In the following we describe two UIMSs using the event description approach.

Menulay

Menulay is an early UIMS using the screen layout as the basis for the dialog model. The UI designer starts by designing the user screen to consist of a set of icons. A semantic action is specified for each icon. This action is performed when the icon is selected. The interface consists of a set of screens. The system generates a set of icon tables giving the name and description of an icon, and a list of (event, function_id) pairs indicating the application function to be called when an event is selected.

Hypercard

This UIMS from Apple incorporates object orientation in the event oriented approach. A *card* has an associated screen layout containing buttons and fields. A button can be selected by clicking the mouse on it. A field contains editable text. Each card has a specific background, which itself behaves like a card. Many cards can share the same background. A hypercard program is thus a hierarchy of cards called a *stack*. UI behaviour is specified by associating an action, in the form of a HyperTalk script, with each button, field and card. The action for an event is determined by using the hierarchy of cards as an inheritance hierarchy. Hypercard uses an interpretive schematic to implement a UI.

EXERCISE 8.5

1. A library information system is to provide three functions—
   (a) Query the issue status of a book
   (b) Search the catalog by author and title
   (c) Issue or return a book.

Design a set of screens for these functions. Visualize the design of a UI using a Menulay or Hypercard like system.

BIBLIOGRAPHY

Kernighan and Plauger (1981) discuss various software tools based on the program preprocessing approach. Miller (1979) is a tutorial on software tools. Text editors are widely discussed in the literature. Meyrowitz and Dam (1981) is a comprehensive survey article on text editors. Myers (1979), Miller and Howden (1981) and Beizer (1983) are important sources on various aspects of program testing. Myers (1979) also describes software tools for program testing. Huang (1978) discusses fundamentals of program instrumentation for software testing. Ingalls (1972), Arthur and Ramanathan (1981) and Power (1983) describe use of instrumentation in different kinds of program analysis. Howden (1975), Koster (1980), Voges (1980) and Woodward and Hedley (1980) discuss aspects of test data generation. Prywes *et al* (1979) is an important article on automatic program generation. Martin (1982) describes generation of application programs. Allison (1983) discusses syntax directed editing. Barstow (1983) and Hunke (1981) are important sources on integrated programming environments.

(a) Software tools in general

1. Kernighan, B.W. and P.J.Plauger (1981): *Software Tools in Pascal*, Addison-Wesley, Reading.
2. Miller, E.(ed.) (1979): *Tutorial: Automated Tools for Software Engineering*, IEEE Computer Society Press.
3. Reifer, D.J. and S. Trattner (1977): "A glossary of software tools and techniques," *Computer*, **10** (7), 52-61.
4. Rochkind, M.J. (1975): "The source code control system," *IEEE Transactions on Software Engineering*, **1** (4), 364-370.

(b) Text editors

1. Embley, D.W. and G. Nagy (1981): "Behavioral aspects of text editors," *Computing Surveys*, **13** (1), 33-70.
2. Engelbart, D.C. and W.K. English (1968): "A research centre for augmenting human intellect," *AFIPS SJCC*, 33, 395-410.
3. Fraser, C.W.(1980): "A generalised text editor," *Commn. of ACM*, **23** (1), 27-60.
4. Macleod, I.A. (1977): "Design and implementation of a display oriented text editor," *Software – Practice and Experience*, **7** (6), 771-778.
5. Meyrowitz, N. and A. Van Dam (1982): "Interactive editing systems," Parts I and II, *Computing Surveys*, **14** (3), 321-352 and 353-416.

(c) Program testing and debugging

1. Beizer, B. (1983): *Software Testing Techniques*, Van Nostrand, New York, 1983.
2. Fairly, R.E.(1979):"ALADDIN: Assembly language assertion driven debug interpreter," *IEEE Transactions on Software Engineering*, **5** (4), 426-428.

3. Foster. K.A.(1980): "Error sensitive test cases analysis," *IEEE Transactions on Software Engineering*, **6**, 258-265.
4. Howden, W.F. (1975): "Methodology for generation of software test data," *IEEE Transactions on Computers*, **24** (5), 554-559.
5. Miller, E. and W.E. Howden (eds.) (1981): *Tutorial: Software Testing and Validation Techniques*, IEEE Computer Society Press.
6. Myers, G.J. (1979): The Art of Software Testing, Wiley, New York.
7. Tratner, M. (1979): "A fundamental approach to debugging," *Software – Practice and Experience*, **9** (2), 97-100.
8. Voges, U. *et al* (1980): "SADAT: an automated testing tool," *IEEE Transactions on Software Engineering*, **6** (3), 286-290.
9. Wilcox, T.R. *et al* (1976): "The design and implementation of a table driven, interactive diagnostic programming system," *Commn. of ACM*, **19** (11), 609-616.
10. Woodward, M.R. and D. Hedley (1980): "Experience with path analysis and testing of programs," *IEEE Transactions on Software Engineering*, **6** (3), 278-286.

(d) Program instrumentation and Performance enhancement

1. Allen, F.E. and J. Cocke (1972): "A catalogue of optimizing transformations," in *Design and Optimization of Compilers*, R. Rustin (ed.), Prentice-Hall, Englewood Cliffs.
2. Arthur, J. and J. Ramanathan (1981): "Design of analyzers for selective program analysis," *IEEE Transactions on Software Engineering*, **7** (1), 39-51.
3. Dhamdhere, D.M., K.S. Shastry, C.V. Ravishankar and N.P.S. Bajwa, (1979): *IIT-FORT User Manual*, Publications Division, IIT Bombay.
4. Huang, J.C.(1978): "Program instrumentation and software testing," *Computer*, 11 (4), 25-33.
5. Ingalls, D. (1972): "The execution time profile as a programming tool," in *Design and Optimization of Compilers*, R. Rustin (ed.), Prentice-Hall, Englewood Cliffs.
6. Power, L.R.(1983): "Design and use of a program execution analyser," *IBM Systems Journal*, **22** (3), 271-294.

(e) Program generators

1. Martin, J.(1982): *Application Development Without Programmers*, Prentice-Hall, Englewood Cliffs.
2. Prywes, N.S., A. Pnueli and S. Shastry (1979): "Use of nonprocedural specification language and associated program generator in software development," *ACM TOPLAS*. **1** (2), 196-217.

(f) Structure editors and Programming environments

1. Allison, L. (1983): "Syntax directed program editing," *Software – Practice and Experience*, **13**, 453-465.
2. Barstow, D.R., H.E. Shrobe, and E. Sandewall (1983): *Interactive Programming Environments*, McGraw-Hill, New York.
3. Fischer, C.N. *et al* (1981): "An introduction to release I of editor Allan Poe," *Computer Science Technical Report 451*, University of Wisconsin.

4. Hunke, H. (1981): *Software Engineering Environments*, North Holland Publ. Co., Amsterdam.
5. Shapiro, E. *et al* (1981): "PASES – A programming environment for Pascal," *SIGPLAN Notices*, **16** (8), 50-57.
6. Teitelman, T. and T. Reps (1981): "The Cornell Program Synthesizer – a syntax directed programming environment," *Commn. of ACM*, **24** (9), 563-573.

(g) User Interfaces

1. Barth, P.S. (1986): "An OO approach to graphical interfaces," *ACM transactions on Graphics*, **5** (2), 142-172.
2. Hayes, F., N. Baran (1989): "A guide to GUI", *Byte*, July 1989, 250-257.
3. Larson (1991): *Interactive software: Tools for Building Interactive User Interfaces*, Prentice-Hall, Reading.
4. Mayhew (1992): *Principles and Guidelines in Software User Interface Design*, Prentice-Hall, Reading.
5. Olsen, D.R. (1992): *User Interface Management Systems*, Morgan Kaufmann, San Mateo, California.
6. Pfaff, G.E. (1985): *User Interface Management Systems*, Springer Verlag.
7. Schneiderman, B. (1983): "Direct manipulation : A step beyond programming languages," *Computer*, **16** (8), 57-69.
8. Schneiderman, B. (1993): *Designing the User Interface*, (2^{nd} edition), Addison-Wesley, New York.