A STRUCTURED APPROACH TO TEACH THE CONTENT OF CS1 TO NON-MAJOR STUDENTS WITH AN EMPHASIS ON ANALYTIC PROBLEM SOLVING

by

MICKEY D. HANEY

B.S., Peru State College, 1982

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1987

Approved by:

[illegible]

Major Professor

DEDICATION

I dedicate this work to my wife Kathy who supported this effort financially and with never ending patience and love. Also to our first child, Cecilia or Glenn, who gave support by kicking Kathy while in the womb.

ACKNOWLEDGEMENTS

Thanks to Dr. Wallentine, Dr. Unger, and Dr. Melton for the support of this project. A special thanks to the GTA's that were patient with the change in method that occurred in the implementation of this papers' content. Also thanks goes to the secretaries in the computer science office for their motherly guidance and technical support. Lastly, a large thank you goes to Rhonda, my lawyer sister, for her weekends of driving to see her brother to edit this paper.

Table of Contents

CHAPTER 1
INTRODUCTION: PROBLEM IDENTIFICATION AND GOAL

The expansion of the availability of micro-computers in primary and secondary institutions has created an interesting problem for the mathematics, business, and science teachers in American schools. As schools have begun to implement computer science programs into their curriculums, it usually has been the mathematics, business, or science instructors who have been asked to teach the newly created courses - often without the benefit of much formal training in this subject much less training in the methods and concepts of how best to teach it. To compound this problem of inexperience, appropriate textbooks are often unavailable. Thus the average instructor of computer science in the public schools is left at a definite disadvantage as he or she prepares to teach the subject matter.

I faced these same problems four years ago when I accepted a full-time job as a high school computer science instructor. As a recent college graduate with a major in mathematics, I had taken computer programming courses but no courses to prepare me to teach in this subject area. I discovered in my initial experience in teaching computer science that a large percentage of the students in my classes failed to grasp the overall concept of programming. This failure of the students reflected my inadequacy in teaching formal analytic problem solving. Briefly, formal analytic problems can be defined as those problems where the preconditions and the desired postconditions are given, and the student is asked to define the translation actions. As applied to computer programming, the student is given the input description and the desired result, and he or she is asked to generate the language instructions to perform the translation.

I found in my classes that if I gave my students a problem where they were to process an input file and produce a report, many were unable to specify the problem in terms of the basic processing configuration of the input file, internal memory, and the output file. In addition, even if some students understood the problem in terms of this configuration, many were unable to define the steps at the language level necessary to solve the problem in the given configuration. This inadequacy in teaching formal analytic problem solving became the motivation for the formulation of the content of this paper.

As a high school instructor I formulated a structured approach for the basic high school computer science course known as Computer Science 1 (hereinafter referred to as CS1) designed to teach my high school students to solve formal analytic computer science problems. Briefly, this approach was designed to teach to specify the pre- and postconditions in terms of a given configuration, to abstract a solution using allowed actions within the configuration, and to implement that abstraction into a high-level language.

In developing this method I utilized the following two basic assumptions :

(1) That the students were initially unable to solve formal analytic problems into a high-level language.

(2) That I lacked a proper approach to teach the student to solve analytic problems.

Most of the high school students would enroll in CS1 because of interest, not because they plan to major in computer science in college. Thus the most important goal in the proposed structured approach to teaching the CS1 course is that of providing the student with an educational experience where his or her analytic skills are developed, not necessarily that of teaching the students the intricacies of a programming language.

The Association for Computing Machinery (hereinafter referred to as ACM) defines the following major objectives for CS1:

- to introduce a disciplined approach to problem-solving methods and algorithmic development.
- to introduce procedural and data abstraction.
- to teach program design, coding, debugging, testing, and using good programming style.
- to teach a block-structured high-level language.
- to provide a familiarity with the evolution of computer hardware and software technology.
- to provide a foundation for further studies in computer science. [1]

The remainder of this report explains a possible course design for CS1 that I developed which allows the above objectives to be met and that also provides a defined method to enhance the high school student's analytic skills in computer science. Chapter Two defines what is meant by a formal analytic problem. Chapters Three, Four, and Five explain the three basic components of the course design, as follows:

(1) A defined problem domain to be used in the creation of analytic prohlems for the student to solve.

(2) A synthetic model for illustrating analytic solutions, for use as a student development model and for providing a formal testing format for analytic problem solving.

(3) Teaching and testing in terms of analytic problem solving.

Finally, Chapter Six will provide conclusions concerning the proposed course design.

CHAPTER 2
DEFINITION OF FORMAL ANALYTIC PROBLEM SOLVING

The goal of this paper is to define a course in which the introductory students can enhance their abilities to specify a problem, abstractly design a solution, and implement that design into a high level language. In other words, the course is designed to develop the students' abilities to solve formal analytic problems. This chapter provides a definition of formal analytic problems.

2.1 Definitions of Formal Analytic Problems and Formal Synthetic Problems

There are two basic types of problems that can arise for people to solve - informal problems and formal problems. Computer science is concerned with the solving of formal problems. "Formal problems are characterized by complete specifications: precisely specified initial conditions, as well as solutions or results of a specified form; and they must be solved by a completely specified set of actions."[2] From this definition is derived the three components of a formal problem as follows: 1) The initial conditions (preconditions); 2) the desired result (postconditions); and 3) the actions to perform the translation (algorithm or program). If all three components of the formal problem are specified, then a complete specification of the problem is present. However, when one of the three necessary components is missing, we have a formal problem that needs solved. Two types of formal problems can be created depending on which one of the components is not specified - problems of synthesis and problems of analysis.

In problems of synthesis the student is given the preconditions and the actions to obtain the desired result. The student normally knows the general form of the desired result but using the preconditions must perform the actions to instantiate the desired result. Following a cooking recipe involves solving a problem of synthesis. In computer science, for example, the student could be given a set of code (actions) and an input list (preconditions) for that code to process, and the student is then asked to specify the output (postconditions). One method to be used in solving synthesis problems is to have available an input list, an empty memory area, and an empty output area. The student then executes the instructions making the changes to the respective areas. The resultant solution of the synthetic problem is the state of the output file at the termination of the code.

In problems of analysis the student is given the preconditions and the desired postconditions. The student must then develop the actions to make the translation from preconditions to postconditions. An example is where a traveler is in Manhattan, Kansas and wants to drive to McCook, Nebraska. His presence in Manhattan, Kansas is the precondition, and his arrival in McCook, Nebraska is the desired postcondition. The goal of the analytic problem is to derive the actions to make the translation. In computer science, for example, the student is given a programming problem which contains the input description (preconditions) and desired result (postconditions), and the student is asked to derive the program instructions to make the translation.

Solving a formal analytic problem depends greatly on the physical configuration and the allowable procedural abilities. For instance, using the trip to McCook example, if the mode of transportation is a car, the physical configuration becomes the highway system and the allowable procedural actions become driving down a highway and turning left or right. However, if the mode of transportation is an airplane, the physical configuration is based on the available air routes and the allowable actions are ascent, descent, and the respective turns during the flight.

CHAPTER 3
DEFINED PROBLEM DOMAIN

As explained in the introduction to this paper, students in CS1 initially have trouble specifying a problem, abstracting a solution, and implementing the solution into a high-level language (i.e., formal analytic problem solving). In order to overcome this problem for the student, the first component necessary for the teaching of formal analytic problem solving in CS1 is the creation of a defined problem domain. The reason for creating this domain is that in solving any type of problem, not only computer science problems, it is important that the students understand the domain or environment in which they are being asked to solve the problem. For example, if the students are asked to solve the analytic problem of how to travel from Manhattan, Kansas to McCook, Nebraska, they understand because of prior experiences the domain or environment they are asked work within if the the environment includes driving a car on the highway system. But a majority of the students probably could not solve the problem if asked to solve it within an environment or domain that includes the use of an airplane and the air routes because of their lack of understanding of this configuration and the allowable actions within it. Likewise in the teaching of CS1, the students must be provided with a domain that they understand before they can develop solutions to problems within the domain.

The proposed course design provides a defined problem domain which consists of 1) a physical processing configuration (i.e., the highway system in the above described example) and 2) the necessary abstract analytic components consisting of data abstraction (i.e., the cities between Manhattan and McCook and the highway numbers in the above described example) and the allowable procedural abilities necessary to process within the configuration (i.e., start the car, turn left, accelerate, stop, etc. in the above described example). Before attempting to solve analytic problems in CS1, the student must become familiar with this defined problem domain. This chapter explains and illustrates the physical processing configuration and the necessary abstract analytic components encompassing the defined problem domain.

3.1 Physical Processing Configuration

In order to create a defined problem domain within which the student can succeed in solving analytic problems, a physical processing configuration must be defined. The physical processing configuration consists of the logical and physical layout of the data storage components. In other words, the configuration defines a concrete model the student must use to solve assigned problems. The physical processing configuration or environment used in this approach consists of three physical components as follows: 1) Input file of records; 2) global memory for program variables; and 3) a single output file. See Diagram #3.1a.

The first component of the configuration, the input file, is sequential. It can contain records with each record having fields of some elemental data type. The assumption is made that all data in the file is correct in terms of range and ordering. A constraint is also placed on the input file that a single record must be read at one time.

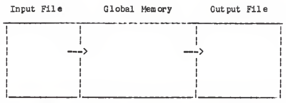


Diagram #3.1a - Physical Processing Configuration

In other words, the separate input of individual fields in a record is not allowed. In addition, the constraint of having one valid record and a sentinel record is placed on the input file.

The second physical component of the configuration is the global memory for program variables. In this component storage locations or variables can be created; and these locations or variables can be given a value using the below described Repetition Dependent Primitives and Non-Repetition Dependent Primitives. In this component the student is able to create memory locations of the basic data types. The component allows random access to the created locations during the execution of the coded module. Upon the creation of a location, it is assumed to contain a 'garbage value'.

The third physical component is the output file. This component is sequential, and its use is constrained to the printing of a whole line at one time. In other words, the printing of part of a line is not allowed at the design level. Output to this component can include the contents of variables and program annotation from the global memory component.

The physical processing configuration as described above provides the processing environment to be used in solving analytic problems. The reason for its induction is that the students are thus given an environment that is constant from problem to problem in which they will be asked to solve problems given the procedural abilities described below along with data abstraction.

3.2 Necessary Abstract Analytic Components

Given the above described physical processing configuration, the abstract analytic components necessary to solve problems within that configuration must be defined. These components include both data abstraction and the procedural primitive abilities necessary to process within the configuration and instantiate the data abstractions.

3.2.1 Data Abstraction

The first component necessary in analysis is the ability to abstract the data in the given problem. ACM defines data abstraction as

> "The conceptual approach of combining a data type with a concomitant set of operations, and the philosophy that such data types can be used without knowing the details of the underlying computer system representation." [1]

However, this definition leaves out an important concept in data abstraction. In abstracting data, the data type, which is "a collection of data values and the definition of one or more operations on those values."[2], must not only be dealt with, but the real world meaning or the semantic descriptor must be recognized as part of the definition as well. A semantic descriptor can be defined as the label given to a particular value that must be instantiated (i.e., a value to be obtained). This semantic descriptor inherits a data type and also includes a textuel statement of the meaning within the real world environment. For instance in the above described Manhattan/McCook example, arrival in the city of Beatrice is the instantiation of a state or step in achieving the end goal of reaching McCook, and the term Beatrice is the semantic descriptor for that desired state. In the sample problem in Chapter Four, the term "call cost" is the semantic descriptor of some real world value. The call cost when instantiated contains e value of the inherent data type, real.

In solving the problems within the defined problem domain, the student must be able to specify the semantic descriptors of the values that are contained in the input file and the semantic descriptors of all values that need to be derived to achieve the desired results of the analytic problem. For example, in the Manhattan/McCook analytic problem, the student must be able to specify the cities or semantic descriptors that need to be achieved or instantiated when driving the route to the final goal of McCook. In the sample problem from Chapter Four the final goal is to produce a report; but to produce the report the 'call cost' of each call must be instantiated and is but one state in achieving the final solution.

3.2.2 Procedural Primitive Abstractions

With the ability to specify the desired semantic descriptors that need to be achieved or instantiated through data abstraction to solve the problem, the procedural primitive abilities or abstractions necessary to instantiate these values and solve problems within the physical configuration must be defined. Within the closure of these primitives, all required abstract semantic descriptors contained within the defined problem domain should be procedurally achievable. Therefore, the remainder this chapter is devoted to defining the allowable procedural primitive abstractions to be used in designing solutions.

3.2.2.1 Processing a Sentinel File Primitive

The first procedural ability that is necessary to solve problems within the physical processing configuration is the ability to process the input file, also known as processing a sentinel file. The students are given the following design construct that presents the logic of inputting one record at a time through the sentinel record:

```
READ FIRST RECORD
WHILE VALID RECORD
     .    .
   <primitives>
     .    .
   READ NEXT RECORD
 ENDWHILE
```

This design construct is used for every problem when processing a sentinel file is required. An example of the use of this primitive is found in Diagram #3.2 using (1) to label the pieces of the primitive.

3.2.2.2 Decision Primitive

The second allowable procedural primitive abstraction is the decision primitive. This primitive provides the ability to control which other allowable primitives, if any, are to be executed based on a relational test of previously defined semantic descriptors and program constants. This primitive provides two basic abilities for the students in relation to semantic descriptors. The first is the ability to expand the boolean semantic information to allow for instantiation of detailed semantic descriptors. For instance, if when necessary to instantiate the semantic descriptor "in-state call count" from the sample problem in Chapter Four, "call" in the semantic descriptor can be utilized through the boolean semantics of the loop condition; but a test must be introduced before we can include the words "in-state" in the semantic descriptor. See Diagram #3.2, (2) labeling.

The second ability this primitive allows is the variation in the method of instantiation of a semantic descriptor. For instance, in order to instantiate "call cost" where the method depends on whether the call cost is in-state or out-state, the decision primitive that asks this question allows "call cost" to be calculated using different methods. The semantics of the value remain constant both ways, but the method of instantiation varies.

3.2.2.3 <u>Repetition Dependent Primitives</u>

Several abstract procedural abilities are based on the repetition of some value. In the proposed problem domain four procedural design primitives are necessary to instantiate inter-record semantic descriptors during repetition. These primitives are: 1) Initialize; 2) Accumulate; 3) Increment; and 4) Select.

The Initialize primitive is used to initially instantiate the semantic descriptors of all of the other repetition dependent primitives. The Initialize primitive is used before the use of the Processing a Sentinel File primitive to illustrate the procedural need to initialize all of the inter-record based semantic descriptors. The Initialize primitive is moved to a high level of abstraction in the design. When designing the solution, all other semantic descriptors are assumed to be initialized through the use of a single Initialize primitive. See Diagram #3.2, (3) labeling. In implementation of this design primitive, the students must find all of the variables associated with the Accumulate, Increment, or Select primitives and instantiate their initial state.

The Accumulate primitive is necessary to derive the summation of some value that occurs multiple times. The Accumulate primitive can only be used within the Processing a Sentinel File primitive. The implications of the use of this primitive in the design are that the summation of some semantic descriptor is instantiated. In design, this primitive is contained within a process box. See Diagram #3.2, (4) labeling. In the implementation of this design primitive, the synthetic form of X = X + Y is used.

The Increment primitive is necessary because of the need to count the number of times some logical state occurs. The Increment primitive can only be used within the Processing a Sentinel File primitive. The implications of the use of this primitive in the design are that the number of times the primitive is executed will be derived. In design, this primitive is contained within a process box. See See Diagram #3.2, (5) labeling. In the implementation of this design primitive, the synthetic form of X = X + 1 is used.

The Selection primitive is necessary because of the need to select the largest or smallest instantiation of a semantic descriptor based on inter-record comparison. The Select primitive can only be used within the Processing a Sentinel File primitive. The implications of this primitive are that the largest or smallest instantiation of some repetitive value is selected. In design, this primitive is achieved through the use of the single sided decision primitive with initialization of the current state, testing of the current state, and update of the current state upon a true relational condition. See Diagram #3.2, (6) labeling. In the implementation of this design primitive, the following synthetic form is used:

```
IF <this_record_state [> <] current_state> THEN
     current_state <-- this_record_state
ENDIF
```

3.2.2.4 Non-Repetition Dependent Primitives

The remaining abstract procedural primitives are not repetition dependent. Therefore they can be used anywhere in a solution design. In the proposed defined problem domain five procedural design primitives are necessary to instantiate intra-record semantic descriptors or semantic descriptors that can be instantiated without repetition. These primitives are: 1) Read; 2) Print; 3) Calculate; 4) Set; and 5) Build and Extract.

The Read primitive is necessary to transfer one complete record from the input file component to the semantic descriptors in the global memory component of the physical processing configuration. The implications of this primitive are that a complete record is transferred to the global memory component to the respective semantic storage locations for later use, and the intrinsic pointer in the input file is advanced to the next record. In design, this primitive is included within an I/O flowchart box. See Diagram #3.2, (7) labeling. In implementation of this design primitive, the synthetic form of READ <variable list> is used.

The Print primitive is necessary to transfer information from the global memory component to the output file component. The implications of this primitive are that at least one line of formatted output is produced, and the intrinsic pointer in the output file is advanced to the next empty line. In design, this primitive is included in an I/O flowchart box. See Diagram #3.2, (8) labeling. In implementation of this design primitive, the synthetic form of PRINT <variables and annotation> <formatting> based on the respective implementation language is used.

The Calculate primitive is necessary to mathematically instantiate a semantic descriptor within the global memory component when the procedural ability to instantiate the semantic value does not require repetition. The implications of this design primitive are that a semantic descriptor has been instantiated through some arithmetic operation(s). In design, this primitiva is included in a flowchart process box. See Diagram #3.2, (9) labeling. In the implementation of this design primitive, the synthetic form of X = <arithmetic expression> is used.

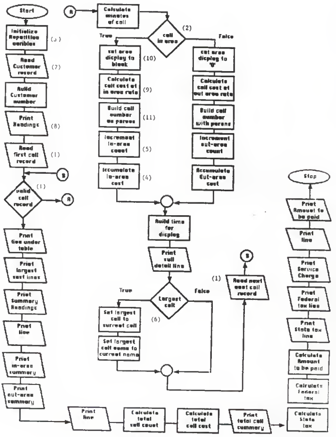
The Set primitive is necessary to instantiate the state of some semantic descriptor without the use of arithmetic or string operations. The implications of this design primitive are that the state of some semantic dascriptor is instantiated. In design, this primitive is included in a flowchart process box. See Diagram #3.2, (10) labeling. In the implementation of this design primitive, the synthetic form of S or X = <constant> is used.

The Build and Extract design primitives ara necessary to perform operations on character strings. The selection of the verb is based on the semantics of tha operation. The implications of the use of these primitives are that a string semantic descriptor has been instantiated through operations on other string values. In design, these primitives are included in a flowchart process box. See Diagram #3.2, (11) labeling. In the implementation of these design primitives, the synthetic form of S = <string expression> is used.

3.3 Conclusion

This chapter presents the defined problem domain in which the student will be asked to solve problems. By creating the defined problem domain for the students, they are no longer asked to design solutions to problems at the language abstraction level. Instead they are able to use a concrete computer model to represent the computer system or physical environment and are then able to design the solution to the assigned problems using the allowable procedural primitives at a level of abstraction closer to English sentence representation used in other disciplines.

Diagram #3.2a - Procedural Primitives



CHAPTER 4
ANALYTIC PROBLEM SOLVING PROCESS

In the proposed course design for CS1 once the students understand the defined problem domain they will be asked to solve formal analytic problems within the domain. As explained in Chapter Two, the goal in any analytic problem is to completely specify the problem (i.e., to completely specify the preconditions, the postconditions, and the actions to perform the translation). The students in CS1, as part of the analytic problem solving process as presentad in the course, must specify the preconditions and postconditions of the problem as given in terms of the physical processing configuration. Next they must design a solution to the problem. Lastly, the students must implement the design into a high leval language. The course provides the following six phasas which are designed to lead tha students through these steps to a complete specification of a formal analytic computer science problem:

1) Specification of Pre- and Postcondition Phase;
2) Dasign Phase;
3) Pre-coding Phase;
4) Coding Phase;
5) Testing Phase; and
6) Documentation Phase.

For each of tha above-listed phases of the problem solving process, the inputs to the phase, the desired results from the phase, and the necessary translation algorithms to be used to derive this desired result from the inputs are defined. Thus this model of the problem solving process, in essence, presents a synthetic model for the students to solve formal analytic problems within the defined problem domain.

The above listed six phases are explained in this chapter. In order to clearly illustrate these six components of the model, a sample problem, which is described below, is used. This problem is selected because it utilizes all of the primitive procedural and data abstraction abilities that are within the defined problem domain as presented in Chapter Three.

SAMPLE PROBLEM:

```
                    CS 20x LAB
                   PHONE COMPANY

Givan an input fila that contains the information
 for one customer to the MUW phone company. The fila
 contains a header record with :

  1. the customer area code      '913'
  2. customer number             '4567892'
  3. monthly service charge      16.75

The remainder of tha records are tha phona calls that
 were made during the month by the above customer and
 includes the following information for each :

  1. Area code called         '913'
  2. Number called            '5638976'
  3. Time of the call         '1020'    for 10:20
  4. seconds of the call      340

Create a bill to the customer that prints on the top
 appropriate headings with the customer phone number
 inserting dashes and parentheses in the out area form
 stated below.
For each call record, print on the detail line an '0'
 if the call was out of the customer area code else
 a blank. Then print the number called in following
 forms, the minutes of the call, the time of the call,
 and the charge based on the following table.

 PHONE FORMS  in area  : 563-8976
              out area : (412) 563-8976

 BILLING RATES
              In area   $.20 per minute
              Out area  $.24 per minute
```

Totals to be kept are :

1. Count in-area calls, out-area calls, and all calls.
2. In area call cost, out area call cost
   and total call cost.
3. State tax at 3% of all calls.
4. Federal tax at 4% of all out-area calls.
5. The amount to be paid by the customer.
6. The number and amount of the most expensive call.

4.1 Specification of Pre- and Postcondition Phase

In analytic problems the preconditions (input description) and the desired result (output description) are given, and the student is asked to generate the actions to make the translation. In the specification phase of the model, the goal is to formally define both the preconditions and the postconditions of a given textual problem such as the above described sample problem in terms of the physical processing configuration. This specification is accomplished in preparation for the designing of the translation or the solution to the analytic problem within the defined problem domain.

The terms precondition and postcondition can be defined as follows:

Precondition
"A condition that must be true before the current instruction(s) will do what we want."

Postcondition
"The result of executing an instruction(s)"[2]

"The pre- and postconditions are really the values of all the variables we have used at any point in the algorithm and the current state of the data list and output list."[2]

The specification phase of the model follows these basic definitions. In specifying the problem the students are asked to define the state of all three components of the physical processing configuration before and after the activation of the global module as a whole. The state of the components during the activation of the module is not of concern. Instead, the students must specify the state of the input file, global memory, and output file components before the execution of the module (preconditions). The students must also specify the desired state of these components after the module has been executed (postconditions). This specification of the state of the three components of the physical processing configuration are further explained below through the use of the sample problem.

4.1.1 Specification of Input File Component

--Precondition of the Input File

In solving the sample problem the students must specify the precondition or state of the input file before the execution of the set of code (or soon to be derived actions). A modeling component is needed because the form of the input file will vary from problem to problem. Diagram #4.1a illustrates the proposed modeling component as applied to the sample problem.

This graphical view represents a file which is broken into records and fields. Each field is labeled with a semantic descriptor and the type of information. The last row or record is dedicated to the description of the sentinel record. If the instructor feels it is

Customer area code	Customer number	Service charge	header record	
char length 3	char length 7	real		
Receiver area code	Receiver number	Time of call	Seconds of call	transaction records
char length 3	char length 7	char 4	Int	
'TRASH'	'X'	9	9	

Diagram #4.1a - Precondition of Input File

necessary, the range of the field can be defined.

To derive this component the students are required to to analyze the given textual problem, extract the respective input requirements, and completely specify these requirements in a defined formal model. The model allows for the complete abstract specification of the contents of the file in terms of records, fields, and the termination condition.

The selection of a value is made in one of the fields of the last record to represent the sentinel value. In this case the word 'TRASH' will be placed in the RECEIVER AREA CODE field. The selection of the sentinel value can either be given by the instructor or defined by the student, depending on the management of the data set.

--Postcondition of the Input File

Next the students must specify the state of the input file after the execution of the global module of code. In the environment for this course as explained in Chapter Three, one record at a time must be read from the input file through the sentinel record. The state of the file after the execution of the module within the scope of the elemental problems presented in the course will be trivial. Simply stated, all records will have been read. The students should derive the following postcondition for the input file component:

Postcondition of the Input File
  The file will be read through the sentinel record.

4.1.2 Specification of Global Memory for Program Variables Component

The pre- and postcondition of the global memory component for a global module is based on the abstract view that the binding of the variables to memory locations and the ability to reference these variables only exists during the activation of the global module. Reference to these variables exists only upon activation of the module in a compiled environment and upon use in an interpreted environment such as BASIC. These variables become dereferenced upon completion of the global module.

--Precondition of Global Memory Component

From the above abstract view it can be understood that no variables exist before the activation of the module. Thus in a global module program, the precondition of the global memory component is as follows:

Precondition of Global Memory Component
  No variables can be referenced

--Postcondition of Global Memory Component

After the execution of the module the variables are dereferenced and not accessible in compiled environments. However, in an interpreted environment the variables exist until they are cleared by a system command such as editing the program. The assumption is made that no referance to the variables exists in the global memory component after the execution of the global module.

In actuality both the pre- and postcondition of this component for any problem will remain constant from problem to problem and therefore nead not be specified. For completeness, however, the following textual statement can be included in the specification:

Preconditions of the global memory component
  No variables can be referanced

Postconditions of the internal memory
  No variables can be referenced

4.1.3 Specification of the Output File Component

The third task that the students must perform in the specification phase is to define the pre- and postcondition of the output file component. The major goal in any problem is to input data, process it, and output results. The specification of the pre- and postconditions of the output file will represent the desired effect that the module will have on the output file component.

--Precondition of the Output File Component

The problems for this course are designed to produce results to an output file. The existence of an output file is the necessary precondition throughout all of the problems that the students will solve. Therefore, the major precondition of the global module output file in any problem is as follows:

```
    Precondition of Output File
       An output file exists
```

--Postcondition of the Output File Component

The state of the output file after the execution of the global module is the desired result of the synthesis of the data values through the soon to be designed and coded instructions. This state varies from analytic problem to analytic problem. Therefore, the students must specify a prototype of the output file component to model the desired state of this component after the execution of the global module. The students should derive the prototype in Diagram #4.1b as the postcondition of the output file component for the sample problem.

Problems that are given to the students are normally given in textual sentence based English. In the creation of a prototype such as illustrated above, the students must analyze the assigned problem and extract the requested output values. These requested values must then

```
12345678901234567890123456789012345678901234567890

               MDW PHONE COMPANY

--------------------------------------------------
CUSTOMER NUMBER : %            %  DATE : %       %
--------------------------------------------------
                  ITEMIZED BILL

    NUMBER CALLED     TIME  MINUTES    COST

--------------------------------------------------
%% %            %    %  %    ####    ####.##
      .                 .        .          .
      .                 .        .          .
--------------------------------------------------

YOUR MOST EXPENSIVE CALL WAS TO %           %
  AND COST ####.##

          NUMBER OF
            CALLS        COST
-----------------------------------------
IN AREA     ####        ####.##
OUT AREA    ####        ####.##
-----------------------------------------
SUMMARY     ####       #####.##

STATE TAX               ####.##
FEDERAL TAX             ####.##
SERVICE CHARGE           ###.##
-----------------------------------------
PAY THIS AMOUNT        #####.##
```

Diagram #4.1b - Postcondition of the Output File

be organized into a machine creatable format with proper annotation specifying the semantic descriptors of the output values.

The generation of this prototype has a two-fold effect. First, it forces the students to restate the desired result, and second, it also presents an exact replica of the output to be generated for use in design and implementation by the students. The prototype generated for any problem presents the logical order of processing. For example, the goal in the sample problem is to create this report and by the nature of the output mechanism must be done from the top of the prototype to the bottom.

In any prototype pound signs are used to illustrate variable numeric fields, and percent signs are used to represent the boundary on variable character fields. A reference line is induced to map the columns in the component. The necessary annotation is also included and should be the exact annotation that is produced in the implementation of the solution.

4.1.4 Conclusion to Specification Phase

In summary, the goal of the specification of the pre- and postcondition phase is to specify the pre- and postconditions of the formal analytic problem within the context of the physical processing configuration. The specification of the pre- and postconditions represent the given parts of a formal analytic problem. In actuality, as shown above, only the precondition of the input file component and the postcondition of the output file component vary from problem to problem; all of the other pre- and postconditions remain constant. However, for completeness and to allow the students to better understand the extension of the model to the specification of submodules later in the CS1 course, the students are asked to specify all pre- and postconditions for each analytic problem to be solved. A complete specification of the sample problem would look like Diagram #4.1c and Diagram #4.1d.

INPUT FILE :
Precondition :

Customer area code	Customer number	Service charge	header record	
char length 3	char length 7	real		
Receiver area code	Receiver number	Time of call	Seconds of call	transaction records
char length 3	char length 7	char 4	Int	
'TRASH'	'X'	9	9	

Postcondition :
1. the file will be read through the sentinel record

INTERNAL VARIABLES
Precondition :
1. No internal variables can be referenced
Postcondition :
2. No internal variables can be referenced

Diagram #4.1c - Specification of Sample problem

OUTPUT FILE
Precondition :
1. Output file must exist

Postcondition :

```
12345678901234567890123456789012345678901234567890

                MUW PHONE COMPANY

--------------------------------------------------
CUSTOMER NUMBER : %            %  DATE : %      %
--------------------------------------------------
                  ITEMIZED BILL

    NUMBER CALLED     TIME  MINUTES    COST

--------------------------------------------------
%% %            %    %   %   ####     ####.##
      .                 .        .          .
      .                 .        .          .
--------------------------------------------------

YOUR MOST EXPENSIVE CALL WAS TO %            %
  AND COST ####.##

          NUMBER OF
            CALLS        COST
----------------------------------------
IN AREA     ####        ####.##
OUT AREA    ####        ####.##
----------------------------------------
SUMMARY     ####       #####.##

STATE TAX               ####.##
FEDERAL TAX             ####.##
SERVICE CHARGE           ###.##
----------------------------------------
PAY THIS AMOUNT        #####.##
```

Diagram 4.1d - Specification of Sample Problem (cont)

The remaining five phases to be completed by the students involve the specification of the third part of any formal analytic problem(i.e., the specifying of the actions to perform the translation from precondition to postcondition). The remainder of this chapter explains those five phases.

4.2 Design Phase

In the first phase of the problem solving process called the specification phase, the students must formally define the pre- and postconditions of the given analytic problem within the context of the physical processing configuration. In the second phase called the design phase the goal is for the students to define the translation from the preconditions to the postconditions using the allowable procedural design primitives within the defined problem domain. In the design phase the students accomplish this translation through the use of semantic descriptors and the allowable procedural primitives. The task of the students is to abstract and define the semantic descriptors of the values in the problem and place these semantic descriptors with the procedural primitive that instantiates each descriptor using their own thought processes and the instructor as verification tools. See Diagram #4.2a and #4.2b. Essentially upon completion of the design phase, the students have mapped out the basic solution to the analytic problem.

Students in CS1 generally find the development of the basic steps for solving an analytic problem to be the most difficult part of the course. The design phase forces the student to specify these steps outside the environment of the computer itself in an abstract form before attempting to implement the abstraction using a high-level language. For instance, since certain semantic descriptors are instantiated through accumulation, the students must specify this process within a repetition primitive in a form that represents the proper procedural method of instantiation. The students simply state "Accumulate" along with the associated semantic descriptor.

This approach to design does not use top-down design strategies. The course approaches the design by attacking the whole problem at one level of abstraction. The student must learn to design a global module before decomposition or top-down strategies can be applied.

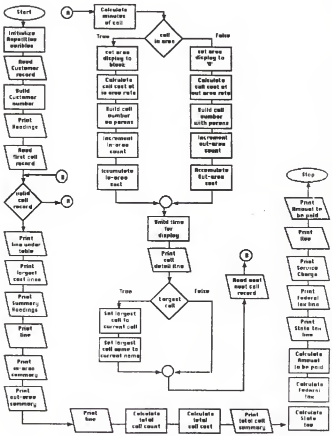
In designing the solution to the problem in the design phase, the students have available two possible forms as follows: 1) Linear Textual form; and 2) Graphical Flowchart form. The two possible design forms of the sample problem are illustrated below in Diagram #4.2a and Diagram #4.2b. The explanation and analysis of reasoning for the design is left to the reader. There is no logical difference between which method is used; however, this author strongly recommends the use of the graphical flowchart form for teaching introductory students because of its graphical nature in nesting decision structures.

Upon careful analysis of the two possible forms, differences can be found in the level of abstraction that was used in the design of the solution to the sample problem. The design phase has no electronic validation method. The level of abstraction that is used is totally up to the instructor teaching the course. For instance, in the graphical flowchart form the step of printing the headings on the top of the report using a single I/O box is defined. This defers the analysis and abstraction of printing all of the lines in the headings to the coding phase. This deference can be construed as a first step in top-down design. In the linear textual form, the printing of each line in the headings is specified and, therefore, is designed at a lower level of abstraction.

Diagram #4.2a - Linear Textual Form of Design

```
BEGIN ALGORITHM
  INITIALIZE ACCUMULATORS AND SELECTORS

  READ HEADER RECORD
  BUILD CUSTOMER PHONE FOR DISPLAY

  PRINT TOP HEADING
  PRINT LINE
  PRINT CUSTOMER DETAIL LINE
  PRINT LINE
  PRINT REPORT TOP HEADING
  PRINT COLUMN HEADING
  PRINT LINE

  INITIALIZE CALL LOOP BY READING CALL RECORD

  WHILE NOT SENTINEL RECORD
      CALCULATE MINUTES OF CALL

      IF CALL IS IN AREA THEN
          SET AREA DISPLAY TO BLANK
          CALCULATE CALL COST
          BUILD CALL NUMBER FOR DISPLAY
          INCREMENT IN AREA COUNT
          ACCUMULATE IN CALL COST
       ELSE
          SET AREA DISPLAY TO 'O'
          CALCULATE CALL COST
          BUILD CALL NUMBER FOR DISPLAY
          INCREMENT OUT AREA COUNT
          ACCUMULATE OUT AREA CALL COST
       ENDIF

      BUILD TIME OF CALL FOR DISPLAY
      PRINT DETAIL LINE

      IF COST OF THIS CALL IS GREATER THAN LARGEST THEN
          SET LARGEST CALL COST TO THIS COST
          SET LARGEST CALL NUMBER TO THIS NUMBER
          ENDIF

          UPDATE CALL LOOP BY READING NEXT RECORD
       ENDWHILE

      PRINT LINE
      PRINT LARGEST COST LINES
      PRINT SUMMARY HEADINGS
      PRINT LINE
      PRINT IN AREA SUMMARY
      PRINT OUT AREA SUMMARY
      PRINT LINE
      CALCULATE TOTAL CALL COUNT
      CALCULATE TOTAL CALL COST
      PRINT TOTAL CALL SUMMARY
      CALCULATE STATE TAX
      CALCULATE FEDERAL TAX
      CALCULATE AMOUNT TO BE PAID
      PRINT STATE TAX LINE
      PRINT FEDERAL TAX LINE
      PRINT SERVICE CHARGE LINE
      PRINT LINE
      PRINT AMOUNT TO BE PAID LINE.
   END ALGORITHM
```

Diagram #4.2b - Graphical Flowchart Form



In the design phase of the problem solving process the students must specify which steps must occur to solve the analytic problem. In the design phase the implementation of each primitive has not been specified, but only that they need to occur.

The students must understand that the design for a given problem cannot be executed by the computer but must be translated into a formal computer language. The correctness of the design cannot be guaranteed by any system checker. The correctness can only be measured by visual inspection and analysis. If the semantic descriptors and the primitives meet the actual needs of the problem, then a correct design has been developed. It is very important to have a visual inspection by an outside source to check the design by the students. If the design is incorrect, then the frustrations are later multiplied. The students should be able to specify on the abstract level what is to be done before the actual implementation is to begin.

After careful inspection of the design, implementation of the design can begin. During the implementation, problems with the design will surface. The inadequacies that are found in the design during implementation should be reflected backward in the model in the design phase. The design component needs to be updated as these problems arise. When the student submits the final solution, all components should be cohesive.

4.3 Pre-Coding Phase

The third phase of the problem solving process is called the pre-coding phase. During this phase the students must predefine all user-defined identifiers which form the data dictionary to be used in coding. The generation of these user-defined identifiers, spacifically, the module name, the input variables, the process variables, and the format variables, have direct synthetic algorithms from the specification and design components which are input to this phase. When the students develop these user-defined identifiers, they must generate a variable name or format list and also determine the proper placement of reference to aach variable within the specification and design components. In this third phase of the problem solving process, the students must for the first time apply the language dependency rules. The generation of the identifiers and format lists must adhere to the language syntax. The variable names that are created during this phase are the only ones that are to be used during the coding of the solution. A complete specification of the variables and formats to be used and their logical positioning in the specification and design components allows for diract translation in the coding phase.

4.3.1 Module Name

The first user-defined identifier in the pre-coding phase is the module name. Once defined, this identifier becomes the name of the program in the coding phase. In top down design this name is derived from the semantic call to this routine. But in the proposad course design a name for the global module is derived by just creating an appropriate name. The name of our sample program will be PHONE_BILL.

4.3.2 Input Variables

The second step of the pre-coding phase is the generation of the input variables from the precondition component of the input file. For every field in this component, a variable of the respective data type must be generated. Diagram #4.3a illustrates the derivation of the input variables in the pre-coding phase of the sample problem.

Customer area code	Customer number	Service charge	
char length 3	char length 7	real	customer record

Receiver area code	Receiver number	Time of call	Seconds of call	
char length 3	char length 7	char 4	Int	call records
'TRASH'	'X'	9	9	

Input Variables

CUS_AREA_CODE$	CHAR 3
CUS_NUMBER$	CHAR 7
SERVICE_CHARGE	REAL
REC_AREA_CODE$	CHAR 3
REC_NUMBER$	CHAR 7
TIME_OF_CALL$	CHAR 4
SECONDS_OF_CALL	INTEGER

Diagram 4.3a - Derivation of Input Variables

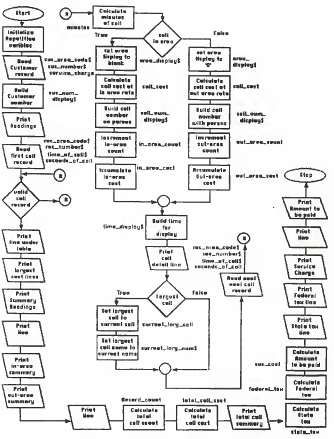
The generation of these variables can be written directly into the program file as documentation or declarations or can be written separately. When the graphical flowchart form is used, the variables should be written beside the respective input box that specifies the record being read for later reference. See Diagram #4.2. When the linear textual form is used, the variables should be placed bes3h. When the linear textual form is used, the variables should be placed beside the respective textual statement that inputs the record.

4.3.3 Process Variables

The third category of variables to be generated are the process variables. One variable is generated for every unique semantic descriptor contained within a process box which begins with the calculate, accumulate, increment, set, build, or extract statements from the design phase. For example in Diagram #4.3b, a process box exists which instantiates the IN_AREA_COUNT, and also a process box exists that instantiates the OUT_AREA_COUNT. The semantic descriptors in these process boxes are different and therefore warrant the creation of a separate variable for each process box. A single variable is created for multiple process boxes when the semantic descriptors are the same. For instance in Diagram #4.3b, one variable is created called CALL_COST for two separate process boxes. This is done because the semantic descriptor in both cases is CALL_COST. In this case, only the method of instantiation varies, not the semantic descriptor of the value. Thus one variable is used for both process boxes.

When the graphical flowchart form is used the variables should, for coding reference purposes, be written beside the respective boxes in the design. The variable should simply take the form of the semantic descriptor in the box. If linear textual form is used, the respective variables should be placed beside or below the primitive that instantiates this semantic value. Below is the generation of the input and process variables using the graphical flowchart form which was input to this phase.

Diagram #4.3b - INPUT AND PROCESS VARIABLE PLACEMENT IN DESIGN



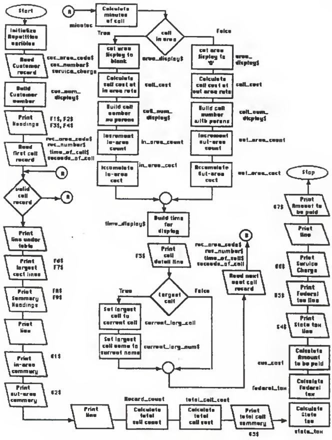
4.3.4 Format Variables

The next step in the pre-coding phase is to develop the formats thet are necessary to format output information. This method varies depending on the lenguage to be used in implementation. If the language uses a graphicel representation, like BASIC, the output picture itself specifies the formats. A varieble needs to be creeted for every line in the output file postcondition model to hold the representetion of the format in the coding. The specification of the format variehles should be placed beside the respective line in the output postcondition that it will format. Also the variahle name cen be placed on the design component for use in the coding phase. These two reference patterns ere shown in Diagram #4.3c and Diagram #4.3d.

```
1234567890123445678901234567890123456789012344567890
              MDW PHONE COMPANY                        F1$

----------------------------------------------------
CUSTOMER NUMBER : %            %  DATE : %       %   F2$
----------------------------------------------------
                ITEMIZED BILL                        F3$

 NUMBER CALLED     TIME   MINUTES    COST            F4$
----------------------------------------------------
# %           %    %   %   ####     ####.##          F5$
----------------------------------------------------

THE MOST EXPENSIVE CALL WAS TO %           %         F6$
    AND COST ####.##                                 F7$

          NUMBER OF                                  F8$
            CALLS       COST                         F9$
-------------------------------------------
IN AREA     ####       ####.##                       G1$
OUT AREA    ####       ####.##                       G2$
-------------------------------------------
SUMMARY     ####      #####.##                       G3$

STATE TAX              ####.##                       G4$
FEDERAL TAX            ####.##                       G5$
SERVICE CHARGE          ###.##                       G6$
-------------------------------------------
PAY THIS AMOUNT       #####.##                       G7$
```

Diagram #4.3c - Format Reference Placement

Diagram #4.3d - Flowchart With All Necessary Variables



In a language such as FORTRAN, formats are numbered, and a numeric list representation is generated. The numbers need to be posted on the input specification and the respective format lists generated. These numbers then need to be placed in the design component by the respective print primitive that will use this format. Diagram #4.3e illustrates the necessary reference and format list generation for the language FORTRAN.

```
12345678901234567890123456789012345678901234567890
               MUW PHONE COMPANY                    1000

-------------------------------------------------- 1005
CUSTOMER NUMBER : %             %  DATE : %     %  1010
-------------------------------------------------- 1005
                  ITEMIZED BILL                     1020

  NUMBER CALLED     TIME   MINUTES    COST          1025
__________________________________________________ 1005
# %           %     %    %   ####     ####.##       1035
      .                  .        .          .
__________________________________________________ 1005

THE MOST EXPENSIVE CALL WAS TO %           %        1040
    AND COST ####.##                                1045

          NUMBER OF                                 1055
            CALLS        COST                       1060
------------------------------------------          1065
IN AREA     ####        ####.##                     1070
OUT AREA    ####        ####.##                     1075
------------------------------------------          1065
SUMMARY     ####       #####.##                     1080

STATE TAX               ####.##                     1090
FEDERAL TAX             ####.##                     1090
SERVICE CHARGE           ###.##                     1090
------------------------------------------          1065
PAY THIS AMOUNT        #####.##                     1100

1000  FORMAT (16X, 'MUW PHONE COMPANY')
1005  FORMAT (49('-'))
1010  FORMAT ('CUSTOMER NUMBER : ', A14, 2X, 'DATE : ', A8)
                  .                  .
```

Diagram #4.3e - FORTRAN Format Reference

Other methods need to be developed for implementation using other languages. The development of the formats is an analysis of the output file postcondition model. Writing of the above-listed structure formats before coding begins can be moved to a dynamic generation of formats during coding when the students become competent in format generation. The rewriting of the formats becomes unnecessary once the skill is developed. But the necessity of specifying the variable or number of each format within the design and output file postcondition model does not change.

4.4 Coding Phase

The solution to the formal analytic problem is developed by the students in the design phase in an abstract form, and in the pre-coding phase the user defined identifiers of the coding phase are specified. The next step in this synthetic model for solving analytic problems requires the students to code the designed solution into the syntax of a high-level language. The word "coding" in this report will refer to the translation by hand, using pencil or pen, of the design into language instructions. Given correct inputs to this phase, the coding becomes a synthetic process.

In the coding of the program the language program structure and the design phase component must be followed directly. For the students to succeed in the translation of the design into code, access to sample programs needs to be available to show the syntax of the translations.

The code is derived from the design and specification components which are now available for analysis. The coding of the sample problem is illustrated below into the language WBASIC. The analysis of the reasoning is left to the reader.

Diagram #4.4a - Coding of Sample Problem

```
900  REM PROGRAM PHONE_BILL
1000 REM  INITIALIZE ACCOMULATORS AND SELECTORS
1010     IN_AREA_COONT       = 0
1020     OUT_AREA_COONT      = 0
1030     IN_AREA_CALL_COST  = 0.0
1040     OUT_AREA_CALL_COST = 0.0
1050     CURRENT_LARGE_CALL = 0.0
1060 !
1070 REM  INITIALIZE FORMATS AND HEADINGS
1080 !
1090 REM  12345678901234567890123456789012345678901234567890
1100 F1$='              MUW PHONE COMPANY'
1110 REM  --------------------------------------------------'
1120 F2$='CUSTOMER NOMBER : %           % DATE : %      %'
1130 REM  --------------------------------------------------
1140 F3$='                     ITEMIZED BILL'
1150 REM
1160 F4$='   NOMBER CALLED    TIME   MINOTES    COST'
1170 REM  __________________________________________________
1180 F5$='  # %           %   #####   ####     ####.##'
1190 REM  __________________________________________________
1200 REM
1210 F6$='  YOOR MOST EXPENSIVE CALL WAS TO %          %'
1220 F7$='   AND COST ####.##'
1230 REM
1240 F8$='            NOMBER OF
1250 F9$='              CALLS          COST
1260 REM  -----------------------------------------
1270 G1$='  IN AREA      ####          ####.##
1280 G2$='  OUT AREA     ####          ####.##
1290 REM  -----------------------------------------
1300 G3$='  SOMMARY      ####         #####.##
1310 REM
1320 G4$='  STATE TAX                  ####.##
1330 G5$='  FEDERAL TAX                ####.##
1340 G6$='  SERVICE CHARGE              ###.##
1350 REM  -----------------------------------------
1360 G7$='  PAY THIS AMOUNT           #####.##
1370 !
2000 REM PROCESSING
2010 !    READ HEADER RECORD
2020      READ CUS_AREA_CODE$, CUS_NUMBER$, SERVICE_CHARGE
```

```
2030 !   BUILD COSTOMER PHONE FOR DISPLAY
2040      CUS_NUM_DISPLAY$ = '(' + COS_AREA_CODE$ + ') ' +
                         STR$ ( CUS_NUMBER$, 1, 3) + '-' +
                         STR$ ( COS_NUMBER$, 4, 4)

2050 !   PRINT TOP HEADING
2070      PRINT F1$
2080 !   PRINT LINE
2090      PRINT RPT$('-', 49)
2100 !   PRINT CUSTOMER DETAIL LINE
2110      PRINT OSING F2$, CUS_NUM_DISPLAY$, DATE$
2120 !   PRINT LINE
2130      PRINT RPT$('-', 49)
2140 !   PRINT REPORT TOP HEADING
2150      PRINT F3$
2160      PRINT
2170 !   PRINT COLOMN HEADING
2180      PRINT F4$
2190 !   PRINT LINE
2200      PRINT RPT$('-',49)
2205 !
2210 !   INITIALIZE CALL LOOP BY READING CALL RECORD
2220      READ REC_AREA_CODE$, REC_NUMBER$, TIME_OF_CALL$,
               SECONDS_OF_CALL
2230 !
2240 !   WHILE NOT SENTINEL RECORD
2260      WHILE (REC_AREA_CODE$ <> 'TRASH')
2270 !       CALCULATE MINUTES OF CALL
2280          MINUTES = SECONDS_OF_CALL / 60.0
2290 !
2300 !       IF CALL IS IN AREA THEN
2310          IF (REC_AREA_CODE$ = COS_AREA_CODE$)
2320 !          SET AREA DISPLAY TO BLANK
2330             AREA_DISPLAY$ = ' '
2340 !          CALCULATE CALL COST
2350             CALL_COST = MINUTES_OF_CALL * .20
2360 !          BUILD CALL NOMBER FOR DISPLAY
2370             CALL_NUM_DISPLAY$ = '      ' +
                                      STR$(REC_NUMBER$,1,3) +
                                      '-' + STR$(REC_NUMBER, 4, 4)
2380 !          INCREMENT IN AREA COONT
2390             IN_AREA_COONT = IN_AREA_COUNT + 1
2400 !          ACCUMULATE IN CALL COST
2410             IN_AREA_COST = IN_AREA_COST + CALL_COST
2420          ELSE
2430             SET AREA DISPLAY TO 'O'
2440             AREA_DISPLAY$ = 'O'
2450 !          CALCULATE CALL COST
2460             CALL_COST = MINUTES_OF_CALL * .24
2470 !          BUILD CALL NUMBER FOR DISPLAY
2480             CALL_NUM_DISPLAY$ = '(' + REC_AREA_CODE$  + ') '
                                      STR$(REC_NUMBER$,1,3) +
                                   '-' + STR$(REC_NUMBER, 4, 4)
2490 !            INCREMENT OUT AREA COUNT
2500               OUT_AREA_COUNT = OUT_AREA_COUNT + 1
2510 !            ACCUMULATE OUT CALL COST
2520               OUT_AREA_COST = OUT_AREA_COST + CALL_COST
2530             ENDIF
2540 !
2550           BUILD TIME OF CALL FOR DISPLAY
2560            TIME_DISPLAY$ = STR$(TIME_OF_CALL$, 1, 2) + ':' +
                                STR$(TIME_OF_CALL$, 3, 2)
2570 !         PRINT DETAIL LINE
2540            PRINT USING F5$, AREA_DISPLAY$, CALL_NUM_DISPLAY$,
                         TIME_DISPLAY$, MINUTES_OF_CALL, CALL_COST
2550 !
2560 !         IF COST OF THIS CALL IS GREATER THAN LARGEST THEN
2570            IF CALL_COST > CURRENT_LARG_CALL
2580 !            SET LARGEST CALL COST TO THIS COST
2590               CURRENT_LARG_CALL = CALL_COST
2600 !            SET LARGEST CALL NUMBER TO THIS NUMBER
2610               CURRENT_LARG_NUM$ = CALL_NUM_DISPLAY$
2620             ENDIF
2630 !
2640 !         UPDATE CALL LOOP BY READING NEXT RECORD
2650           READ REC_AREA_CODE$, REC_NUMBER$, TIME_OF_CALL$,
                    SECONDS_OF_CALL
2660         ENDLOOP
2270 !
2280 !    PRINT LINE
2290          PRINT RPT$('-',49)
2300 !    PRINT LARGEST COST LINES
2310          PRINT USING F6$, CURRENT_LARG_NUM$
2320          PRINT USING F7$, CURRENT_LARG_CALL
2340 !    PRINT SUMMARY HEADINGS
2350          PRINT
2360          PRINT F8$
2370          PRINT F9$
2380 !    PRINT LINE
2390          PRINT RPT$('-', 41)
2400 !    PRINT IN AREA SUMMARY
2410          PRINT USING G1$, IN_AREA_COUNT, IN_AREA_COST
2420 !    PRINT OUT AREA SUMMARY
2430          PRINT USING G2$, OUT_AREA_COUNT, OUT_AREA_COST
2440 !    PRINT LINE
2450          PRINT RPT$('-', 41)
2460 !    CALCULATE TOTAL CALL COUNT
2470          RECORD_COUNT = IN_AREA_COUNT + OUT_AREA_COUNT
2480 !    CALCULATE TOTAL CALL COST
2490          TOTAL_CALL_COST = IN_AREA_COST + OUT_AREA_COST
2500 !    PRINT TOTAL CALL SUMMARY
2510          PRINT USING G3$, RECORD_COUNT, TOTAL_CALL_COST
2520 !    CALCULATE STATE TAX
2530          STATE_TAX = TOTAL_CALL_COST * .03
```

```
2540 !   CALCULATE FEDERAL TAX
2550      FEDERAL_TAX = OUT_AREA_COST * .04
2560 !   CALCULATE AMOUNT TO BE PAID
2570      CUS_COST = TOTAL_CALL_COST + SERVICE_CHARGE
                   + STATE_TAX + FEDERAL_TAX
2580 !   PRINT STATE TAX LINE
2590      PRINT USING G4$, STATE_TAX
2600 !   PRINT FEDERAL TAX LINE
2610      PRINT USING G5$, FEDERAL_TAX
2620 !   PRINT SERVICE CHARGE LINE
2630      PRINT USING G6$, SERVICE_CHARGE
2640 !   PRINT LINE
2650      PRINT RPT$('-', 41)
2660 !   PRINT AMOUNT TO BE PAID LINE
2670      PRINT USING G7$, CUS_COST
2680  STOP
```

The above coding is the translation of the specification, design, and precoding components into the language WBASIC. The coding phase follows the design component directly. The students need to be able to hand code a solution using the necessary keywords from the language and the user defined identifiers from the pre-coding phase. The goal at this point is to have a hand-written program that is close to the desired solution. This coding is now input to the testing phase for debugging and verification.

4.5 Testing Phase

Upon the completion of the coding phase, the solution has been coded into a high-level language. This coding is now submitted to the testing phase of the analytic problem solving process. The testing phase includes the removal of language syntax errors and the verification of the output that is produced with the desired postcondition of the output from the specification phase. This verification includes the correctness of the output values and the physical form of the output.

The first step in the testing phase requires the students to enter the coded program using a system editor. Next the source file should be submitted to the compiler or interpreter for debugging. The result is then analyzed by the students. Based upon their analyses, the system editor is used again to make the necessary changes. The students repeat these steps until the desired postconditions are achieved.

A correctly tested program will contain no syntactical errors and will produce correct output. This phase is frustrating for the beginning students because the debugging of errors takes experience. The students' understanding of syntactical forms and their ability to hand synthesize code are very necessary concepts to support this phase. The availability of the instructor and laboratory help can greatly reduce the students' anxiety in the beginning.

In this report no attempt is made to define the methods in the testing phase. It is only important for the reader to understand the desired result of this phase and its importance in the problem solving process. If careful development of the program is undertaken, the testing phase can be minimized. The magnitude of effort that is required in this phase is inversely proportional to the amount of abstract verification effort in the previous phases.

4.6 Documentation Phase

The final phase in the analytic problem solving process is the documentation of the verified solution. Documentation actually occurs all through the analytic problem solving process. The specification phase involves the documentation of the precondition and postcondition of the formal analytic problem. The design phase involves the documentation of the design of a solution to the problem. The coding phase requires the students to document the instructions or the actions into a high-level language. This coding by hand is normally discarded as documentation because the actual source code that is entered in a file replaces it. However, the student is asked to include additional documentation during the documentation phase.

One form of documentation the students must generate during this phase is the global header block at the top of the program. The form of this documentation can vary based on the instructor's desired form; but a sample form is presented in this report. The inclusion of the documentation is felt to be a necessary component in the final solution for maintenance at later stages in the life of the software solution. Diagram #4.6a illustrates one possible form of the global header block.

Another form of documentation that the student may need to generate is a user manual for the software package. Such a user manual could include a definition of the purpose and use of the package and include a tutorial for entering the data and how to execute the program once the data is entered.

Othar forms of documentation at this point could be internal comments within the program to break up the basic parts of the source code. Tha students can be asked to document to any level of abstraction that is required by the instructor. The important factor is that the students hava access to illustration material which contains the

```
900 REM PHONE BILL
910 !
920 !  PROGRAMMER       : MICK HANEY
930 !  DEVELOPMENT DATE : 11/11/86
935 !  LANGUAGE         : WBASIC
940 !  PURPOSE          :
950 !
960 !      TO PROCESS A PHONE BILL FOR AN INDIVIDUAL USING THE
970 !       MELTON, UNGER, WALLENTINE PHONE COMPANY.
980 !
990 !  EXTERNAL DOCUMENTATION :
900 !
910 !      AVAILABLE IN FOLDER - PHONE BILL
920 !
930 !  DATA DICTIONARY
940 !
950 !   INPUT VARIABLES
960 !     CUS_AREA_CODE$......area code of customer
            .       .            .
```

Diagram #4.6a - Possible Global Header Block

desired form and level of ebstrection in the documentetion.

4.7 Conclusion of Analytic Problem Solving Process

The ebove analytic problem solving process defines the steps necessary to completely specify the solution to a formal analytic problem. The students were asked to specify the pre- and postconditions of a given analytic problem end then specified the actions to perform the translation. The introduction of this defined process provides three hesic benefits in the course design. First, it provides the instructor with an illustration method for the development of analytic solutions. Second, it provides the student with a defined synthetic process to use in solving essigned problems. Third, it provides e format for complete testing of analytic problem solving ability.

CHAPTER 5
IMPLEMENTATION OF THE COURSE

In Chapter One two basic assumptions were asserted. First, the students were unable under more traditional methods of teaching CS1 to solve formal analytic problems. Second, the instructor lacked a method to teach these students to solve such problems. The goal in this chapter is to define a course design that concentrates on the teaching and student development of solutions to analytic problems. Briefly, the first step in the course design is to teach the students the defined problem domain. In addition, the development of analytic solutions is illustrated. Also, the students are asked to solve problems within the defined problem domain. This chapter is devoted to describing a course which concentrates on illustrating the complete development of analytic solutions both in presentation by the instructor and development by the students. The three basic phases of the proposed CS1 course are explained in detail below. In addition a formal method of testing the comprehension of analytic skills is described.

5.1 Phase One

The first phase in the proposed course design includes the following major objectives:

(1) An overview of the system that the student will use in testing solutions to analytic problems;

(2) An overview of the defined problem domain including the physical processing configuration and reverse analysis of the necessary abstract analytic components; and

(3) An overview of the language to be used in implementation of analytic solutions.

The first major objective of this phase in the course is to teach the students to use the computer system to be used in the testing phase of tha analytic problem solving process. For example, the students should be taught to logon tha system, execute the necessary operating system commands, and use the necessary tools such as editors and compilors. Tha students should parform a lab assignmant that tests the achievement of this objective.

The second major objactive is the teaching of the defined problem domain consisting of the physical processing configuration and the necessary abstract analytic components. The method of teaching the defined problem domain is chalkboard execution. Further explanation of this method is contained in the appendices. In this method the lab assignment program that was entered and axecuted for teaching the system objective should be used. This set of code is hand executed on a chalkboard by the students as a class to illustrate both the physical processing configuration and tha procadural and data abstraction abilities within tha configuration. After board executing the set of code several times, the students begin to derive through instructor guidance the procedural abilities through reverse analysis. For instance, the students will see aftar several board executions that a statement of the form X = X + Y used within a loop accumulates the value Y. Thus through reverse analysis, the abstract design primitives are derived, and also the concept of data abstraction and instantiation is developed.

The third major objective of this phase is to give the students an overview of the high-level language to be used in implementation of analytic solutions. This overview should illustrate the program structure, statements, and the respective syntax that is associated with these statements. If the lab assignment program utilized to teach the system objective is prepared so that all statements that are necessary at this level of programming are used, the instructor has access to a strong aid for illustrating program structure and the statements within the program. A large amount of time should not be spent on memorization of syntax. The goal is to solve analytic problems, and the learning of syntax is inherent through using the language as a final implementation of the solution and availability of sample code.

After the completion of the first phase, the students have experienced the entering and executing of a program in a certain language and in a certain system environment. The students should also have an abstract view of how the program works in terms of the input file, output file, and global memory components of the physical processing configuration. The students should begin to think of data in terms of the actual instantiation and the abstraction of data into semantic descriptors. In addition, the students should be able to understand the nature of the procedural abilities necessary to solve analytic problems. Lastly the students should have an understanding of the general form of a program and the use of statements in that program.

5.2 Phase Two

The first phase presented the students with an understanding of the computer system, the defined problem domain, and an overall view of the language that is to be used in implementation. In this second phase the students begin to solve formal analytic problems. The objectives of this phase are as follows:

(1) To introduce the students to the analytic problem solving process;

(2) To teach the use of certain procedural primitive abilities introduced during the first phase of the course; and

(3) To use the analytic problem solving process to solve formal analytic problems containing the procedural primitive abilities referred to above.

The introduction of the analytic problem solving process described in Chapter Four is the first objective in this second phase. The students need to understand what an analytic problem is and the method that will be used to solve these problems. A handout illustrating the workings of the analytic problem solving process should be made available to the students.

The second major objective in phase two are the teaching of the concepts of sentinel file input, formatted file output, and internal arithmetic processing. The file input requires the concept of repetition using the sentinel value technique. The arithmetio processing includes the primitive concepts of calculation based on one record, accumulation, increment, summary calculation, and formatted output using the print primitive. Using board execution in the previous phase should already have introduced the student to these concepts.

The third major objective in this phase is for the students to be able to develop a solution to an analytic problem that contains the above concepts using the formal analytic problem solving process. The solving of an analytic problem should illustrate the desired conceptual primitives and the ability to formally derive the solution to this problem. The method suggested is simply for the instructor to present an analytic problem that contains the desired concepts and to use the above synthetic model to solve the given problem. In working this problem, the illustration of all reasoning for the development needs to be provided by the instructor. Performing the above task illustrates the development of a solution to an analytic problem using the problem solving process and also illustrates the desired procedural primitives and data abstraction.

The testing within this phase should be the assignment of an analytic problem that contains the same concepts that were illustrated in the instruction example. The students should be forced to use the problem solving process to solve the assigned problem. Each step in the process needs to be verified by the instructor before the student is allowed to continue. Several class periods can be dedicated to the verification by the instructor of each individual phase of the process. The goal of the lab is to implement solutions to analytio problems. The solution is not just the program. It also includes the specification of pre- and postconditions and the derivation of the actions.

The result of this phase is the exposure of the student to the complete development of two analytic solutions that contain the desired procedural primitives, data abstrection, and the use of a computer system for verification of that solution. Remember that the teaching of the solving of analytic problems was stated in Chapter One as one of the major factors necessary in the teaching of CS1 students; and using the above method illustrates twice the development of analytic solutions.

5.3 Phase Three

The second phase illustrated the analytic problem solving process, the procedural primitive abilities including file processing, formatted output, and arithmetic abilities, and data abstraction. In this third phase the ability to make decisions as to what processing is to occur is added to the concepts presented in phase two. The objectives of this phase are as follows:

(1) To introduce the students to the decision primitive explained in 3.2.2.2 of this report;

(2) To introduce the students to the selection primitive explained in 3.2.2.3 of this report;

(3) To introduce the students to the Set, Build, and Extract design primitives explained in 3.2.2.4 of this report; and

(4) To introduce the students to the nesting of the decision primitive.

The first objective, which is the introduction of the decision primitive, allows the student to control which other primitives are to be executed. The result is that the students can now vary the method of instantiation of a semantic descriptor and also can now instantiate more detailed semantic descriptors. For instance, the students can now instantiate the call cost using different rates and can count both the in-area and out-area calls in the sample problem in Chapter Four of this report by using a decision structure.

The second major objective of the third phase, which is the introduction of the selection primitive, allows the students to select the largest and smallest instantiation of a semantio descriptor during repetition. This primitive is designed using the decision primitive and therefore is procedurally achievable after the decision structure has been introduced.

The third objective of this phase is the introduction of the Set, Build, and Extract procedural primitives. These primitives allow the students to perform operations on character data. The operations on character data include the expanding or extraction of a string of characters to instantiate a new string of characters.

The last major objective in this phase is to teach the student to nest decision structures. This concept expands the use of the decision structure to multiple use of the structure within the true or false side of a decision structure.

The implementation of this phase is done in four parts. The instructor will introduce an analytic problem that contains a single decision structure and uses the analytio problem solving process to derive the coded solution. The students are then assigned a similar analytic problem and asked to use the analytic problem solving process to derive a tested and documented solution. Next this process is repeated using analytic prohlem which requires the nesting of decisions.

The results of the third phase is the students exposure to the procedural concepts from the previous phases with the addition of the decision structure and the detailed generation of values that are associated with the decision structure. The concepts of selection and string processing have been introduced. The students have now been exposed to the complete development of the solution to six analytic computer problems and formal testing of transfer of this ability should be occur.

5.4 Testing

The above phases cover one half of a semester to teach the content of CS1. At this point in time a midterm test is induced to test the competence of the students in developing solutions to analytic problems within the defined problem domain form Chapter Three.

The method suggested for testing comprehension of the primitive concepts and the students ability to solve analytic problems is to induce an analytic problem which necessitates the use of all analytic abilities in the problem domain and the derivation of the solution using the analytic problem solving process. The form of the test should be available to the student at the beginning of the semester for viewing and proper preparation. The suggested time limit is two hours.

5.5 Conclusion

The above phases present a possible implementation of a CS1 course which concentrates on the complete derivation of analytic solutions.

Through the implementation of the above course design the students are exposed to six complete derivations to analytic problems within the defined problem domain. Three of these exposures were illustrated by the instructor and three by the students in preparation for formal testing. The phases are progressive in terms of problem complexity and follow the content of the text [3].

CHAPTER 6
CONCLUSION

The problem that this paper addressed, as defined in chapter one, was to create a learning environment where the average high-school student could succeed in solving formal analytic problems in the course CS1. To create this environment three basic components were introduced. First, a problem domain was defined which included the physical processing configuration and the abstract analytic abilities necessary to solve problems within the domain. Second, an analytic problem solving process was defined for consistent instructor illustration and consistent student development of analytic solutions within the defined problem domain. Third, a course design was defined which concentrated on illustration, development, and testing of the complete process of solving analytic problems.

The above components when combined together present a course design which structures the introductory students' initial experience in solving formal analytic problems in computer science. Upon completion of the course design in this paper the students need to be introduced to top-down design. Further research is necessary in this method to extend the problem domain, the analytic problem solving process to include top-down design for decomposition of more complex analytic problems, and the course design to support these necessary objectives.

References

[1] Koffman, E. and Miller, P. and Wardle, C. (1984). "Recommended Curriculum for CS1, 1984", Communications of the ACM, Oct 1984, vol 27, Num 10, pp 998-1001.

[2] Wetzel, G. and Bulgren, W. (1985). The Algorithmic Process, An Introduction to Problem Solving, Chicago, IL.: SRA, Inc.

[3] Shelley, Gary B., and Cashmann, Thomas J. (1982). Introduction to BASIC Programming, Brea, CA.: Anahiem Publishing Co.

APPENDIX A
CHALKBOARD EXECUTION

The first phase of the course in this paper refers to a method which was called chalkboard execution. This appendix is a definition of, that method. The purpose of the method is to illustrate the separation and interaction of of the physical processing configuration, to illustrate the synthesis of data through the physical processing configuration, and to allow for reverse analysis of the abstraot analytio abilities.

INITIALIZATION

The necessary physical tools are the availability of a chalk board and a common set of high-level source code that contains the desired processing concepts. The board is divided into three separate areas representing the physical processing configuration. One board represents the input file component, one represents the global memory component and one board represents the output file component.

The input data is initially placed in the board section dedicated to the input file component. The data should be in the same form that is needed in the specific language. An arrow is positioned before the first item in the input file and represents the intrinsic file pointer. After each successive field is read this data pointer is moved to the next field of data. The postcondition of the input file component in the specified environment states that all records will have been read and the data pointer should end up after the sentinel record.

The global memory component board is initially cleer because before the ectivation of the module of code no reference to the internal memory exists. At the completion of the module the memory area should be erased because at the termination of the global module all reference to internal memory is lost.

The output hoerd is initially cleer also. The precondition of the output file is simply that the file exists and we assume thet the file is empty. The postcondition of the output file will contain the output from the module that is to be synthesized using the deta in the input file.

BOARD EXECUTION

Given the initial board setup, the common set of code is to be synthesized on the board by the students performing one elementel function per person and moving the responsibility around the room. Verificetion of actions that are performed by the students is necessary hy the instructor and guidance is required when necessary. At criticel points in the execution the instructor can discuss the present state of the synthetic process.

We begin by starting at the top of the program and executing instructions in the logical order thet the computer would use. In a compiled lenguege the first section of code will be the declaration of variables. When this is errived at in the code each student should draw a box in the memory eree end label the hox outside with the name and type of the location. The box should not contain a value because the creetion of a location does not necessarily define it. Variable gain a value initially when used in a read statement or on the left hand of an assignment statement.

In an interpreted environment the variables are not created until the variable is used in the code. The creation of variables in this environment occurs when the student goes to execute his instruction and there exists no variable in the global memory component that is to be defined. In the interpreted environment the student performs both a variable creation and an instruction execution.

We then move to the execution of the ectual instructions in the body of the code. The read stetement takes the next value in the input file and places it in some variable in the memory section. The data pointer is advanced by this same student to the next item in the input file. In reading a record I let one student read one field end not the whole record. The importance of this is that the student sees that the computer reads the deta one field at a time end an error can occur in the middle of a read statement.

The write statement tekes values from variables in the global memory component along with program annotation and transfers them to the output file component. The implementation again here is to allow one student to transfer one field to the output to show that an error cen occur in the middle of a write statement. At this time the formatting in the language cen be discussed. This discussion can also be deferred to the second session of board execution.

The arithmetic statements use variables in the memory and program constants to assign a value to another variable in the memory area. If arithmetic is necessary a part of the memory board is dedicated the ALU in which the student must perform the computation. The student should perform the computation and place the result in the variable that is assigned.

We do not represent effects by conditional statements on the board. Each time we arrive at one, the teacher discusses the condition in terms of the present state of the memory component and the class decides whether it evaluates to true or false. Upon this decision the next student executes the respective statement depending on the condition.

STOP STATE

At the termination of the module the last thing to occur is the erasing of the internal memory by a student. This represents the abstraction that the variables in memory are dereferenced upon the termination of a module. The data pointer in the input file should be pointing past the sentinel record. The only thing that remains that is different from the precondition of the board is the output that is generated in the output file.

CONCLUSION AND CONCEPTS

The use of this method is further enhanced if the code used is the same assignment that the student used in the system exercise. The student has entered and executed this set of code and achieved the output. The output on the board should be the same output that was submitted by the student in the lab assignment. This leads the student to a strong understanding what the program that was used in the system lab did.

The concepts that can be illustrated using this method are only limited by the instructors initiative to stop during execution and discuss the states. The student should see the separation of the input file, output file, and internal memory components and gain an understanding of how they interact. They see that a value cannot go directly from the input file to the output file without first being stored in memory. They see how values can be generated using other values and program constants and that this instantiation can be stored in a memory location.

One session of board execution is not felt to be enough to allow the student complete comprehension of the processing environment. On the second time through the student sees the processing again and probably retains more knowledge. The third effort can be done using a different set of code to get rid of the monkey see monkey do aspect.

The method also allow the student to get out of the chair and perform in front of peers. This can induce laughter but can also be frustrating for some students and requires careful guidance and motivation on the instructors part. This is not to say that the instructor cannot bust a student in the chops with a joke or two for performing the wrong task.

A STRUCTURED APPROACH TO TEACH THE CONTENT OF CS1 TO NON-MAJOR STUDENTS WITH AN EMPHASIS ON ANALYTIC PROBLEM SOLVING

by

MICKEY D. HANEY

B.S., Peru State College, 1982

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1987

ABSTRACT

More and more students are enrolling in introductory programming courses at the secondary level. Tha translation of tha rasponsihility of instructing introductory programming oourses to the secondary lavel has created large problem for secondary instructors who have little formal training in the computer science discipline. This raport is tha result of the efforts by the author to davelop a coursa to taach introductory students to solve formal analytio computer science problems.

The raport defines a problem domain in whioh the students will be asked to solve formal analytic problems. This problem domain consists of a physical processing configuration and the necessary ahstract analytic components to solve problems within the physical processing configuration. Next tha report defines an analytic problem solving process which is used to completely specify the solution to formal analytic problems within the defined prohlem domain. This process includes the specification of the pre- and postcondition of the physical processing configuration and then tha dasign and implemantation of the actions to perform the translation from pre- to postcondition into a high level language.

Lastly, the report defines a possihle implementation of a course which concentratas on illustration and development of complate solutions to formal analytio problems in computer science. Tha course defined in this paper has been implemanted at the secondary level and for two semesters at Kansas State University in the CS200 language lahs.