AN AUTOMATED REGRESSION TESTING SYSTEM FOR COMPILERS AND INTERPRETERS

by

ROBERT L. HORNEY

B.S., Metropolitan 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

1988

Approved by:

[illegible]

Major Professor

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

An important problem facing software developers is how to produce reliable computer software in a cost effective and timely manner. The cost of hardware continues to decrease while software costs rise. As systems become more software-oriented rather than hardware-oriented, the complexity of the software requires that more effort be placed in the areas of software design, development and testing. Software developers must find ways of reducing the costs of software, without affecting its quality, in order to produce a product which is capable of competing in today's marketplace.

Several studies [ALB81, WOL84] have been made concerning the economic costs associated with software development. The results of these studies indicate that the earlier a software error is found, the less expensive the error will be in terms of the overall cost of the product.

Software validation[1] plays an important role in reducing the cost of the product while at the same time assuring the product's quality and reliability. It is used to assure the producer that what the consumer will get meets the intended specifications. The earlier an error is detected in the life cycle of a product, the less expensive it will be for the producer to correct the error.

Compilers play a critical and sometimes overlooked role in determining the success or failure of a project. A faulty compiler may create software errors which can sometimes be hard to find. For this reason, compilers should be thouroughly tested and validated prior to release.

1. The term "software validation" refers to the act of determining the correctness of software with respect to the user's needs and requirements. Software testing, which is one aspect of validation, is the examination of the behavior of the software using sample test cases.

The validation of a compiler for a given language can sometimes be an arduous task. A compiler tends to be a very complex program or set of programs which is capable of accepting a multitude of inputs and producing an object file which may or may not do what was intended. In fact, there are an infinite number of possible programs which can be compiled by any one compiler. Test cases have to be written which will exercise all aspects of a compiler's operation, including the resulting object file.

If the code produced by a compiler contains errors, software programs which make use of the compiler will also contain errors. Testing and maintenance of these programs becomes very difficult because, to the programmer, the source code may appear to be correct. A compiler which is free of critical errors is a necessity for its use as a development tool in the production of software systems.

This paper gives an overview of tools and techniques used for the dynamic validation of software. References are given for tools which are being used or have been used for compiler validation. Methods used in test case selection and coverage are presented along with the phases of software testing which make use of the selected test cases. A tool is presented which can be used for the testing of both compilers and interpreters for various software languages.

1.1 An Overview of Dynamic Validation Techniques

There are many software validation techniques which may be used to demonstrate the correctness of the software which is being tested. Each of these techniques, depending on applicability, may or may not be used during each phase of the software development life cycle. The common goal of these techniques is to demonstrate that the software which is being tested is free from errors.

Boehm [BOE84] describes the software development process in terms of a waterfall model. At the end of each phase in the model, validation is used as a means of assurance that the software (or requirements) is functionally correct and as error free as possible. If enough errors are detected, the software is returned to the beginning of that phase for further work.

Software validation techniques can be broken down into two primary categories [FAI78, MIL81]: static and dynamic. In static analysis, a program is analyzed without regard to its behavior during execution, while in dynamic analysis, a program is analyzed during controlled execution.

Dynamic program testing methodologies include code-based testing (or white box testing) and requirements-based testing (or black box testing). Figure 1 gives a hierarchical diagram of dynamic testing methodologies.

1.1.1 Code-based Testing

Code-based testing involves examining the code within a software module and developing test data and test cases which can be used to exercise that software.

This form of testing can be further subdivided into mutation analysis, symbolic execution, executable assertions, and structural analysis.

Mutation Analysis

Mutation analysis [OST83, POW82, VOG85] *seeds* a program with a number of changes (i.e., errors). A number of *mutant* programs are created with different sets of errors. A set of test cases is then fed as input data to each of the *mutant* programs. The outputs from each program execution are then compared to the outputs of the original (presumably correct) program. If any differences are observed between the outputs, the *mutant* is then discarded. If some *mutants* remain after all programs have been executed, functional analysis is then performed on the remaining programs. If a large number of mutant programs remain, more test data should be added to the original set, with the goal of causing some of the mutant programs to have different

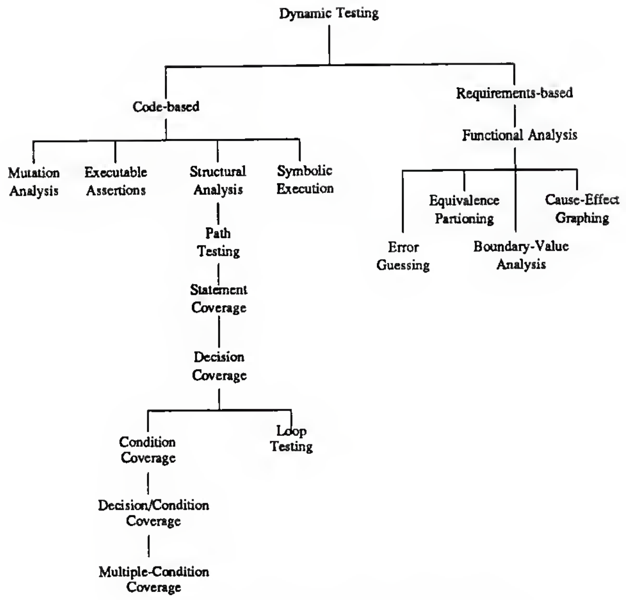

Figure 1. Dynamic Testing Hierarchy

output results from the original program.

Symbolic Execution

Symbolic execution [OST83, SAI84; VOG85] computes the values of a program's variables as symbolic expressions. These expressions represent a sequence of operations performed as the program execution moves along a specific path. For example, if the path leads to an output statement, symbolic execution will show how each of the values in the output statement are computed. An example of symbolic execution is shown in Figure 2.

```
Program:                     Symbolic execution:

READ A,B                     A = a, B = b
C = A + B                    C = a + b
IF C < A THEN D = C          if(a+b) < a then D = (a+b)
    ELSE D = A + C + 1           else D = a+(a+b)+1
C = D + B + A                if(a+b) < a then C = (a+b)+b+a
                                 else C = a+(a+b)+1+b+a
WRITE C                      if(a+b) < a then write a+b+b+a
                                 else write a+a+b+1+b+a
```

Figure 2. Symbolic Execution Example

Executable Assertions

Executable assertions [STU81, POW82] includes code (i.e., assertions) for checking the validity of the values which it computes. The output from a program containing assertion statements is normally a list of the assertion checks and a list of the exception conditions with trace information. Assertion statements are in the form:

```
/* ASSERT condition */
```

This translates to:

```
IF NOT (condition) THEN
        process assertion violation;
```

S. H. Saib [SAI84] describes a method for using assertions in the formal verification of a program. Formal verification attempts to demonstrate that a program operates correctly for all possible combinations of input values. For example, the assertion *assert min* < *0* states that a variable named min should always be less than zero. Assume that a programmer writes the statement *min* = *min* + *1*, when the correct statement should have been *min* = *min* - *1*. A section of a program containing the assertion and the invalid statement might be:

```
assert min < 0;
min = min + 1;
assert min < 0;
```

A theorem is then formed that states *if min < 0 then min + 1 < 0.* The theorem must be true for all values of min in order for the program to be consistent with its assertions. In this example, it is not true for all values of min; therefore, the program is said to be inconsistent with its assertion. If the correct statement had been used, the theorem would have read *if min < 0 then min - 1 < 0.* This theorem is true for all values of min and the program is said to be consistent with its assertions.

Structural Analysis

This form of code-based testing makes use of a program's structure for test-case design and coverage.

Path Testing - Path testing [VOG85, BEI84, MYE79] is a test technique which is used to test a set of paths through a program's structure. A path through a program is any executable sequence of instructions through that program. If all possible paths through a program are tested, and no errors are detected, the program can then be said to be "path tested". The problem with doing this form of "exhaustive" path testing is that the number of individual paths through a program tends to be astronomically large. Other forms of structural analysis have been developed which provide varying levels of coverage.

Statement Coverage - Statement coverage testing [BEI84, MIL84, MYE79] requires every statement in the program being tested to be executed at least once. Statement coverage is often designated by the term *C0*. It can be quantified by the total number of statements in a module that are executed, divided by the total number of statements present in the module.

Decision Coverage - Decision coverage testing [HOW81, MIL84, MYE79] requires each branch from a decision statement to be executed at least once. As an example, the following line of code contains a single decision statement with two conditions, A & B:

```
if(A && B)
```

Valid sets of test cases for decision coverage would be:

```
(1) {(A: true, B: true) and (A: false, B: false)}
    or,
(2) {(A: true, B: false) and (A: true, B: true)}
    or,
(3) {(A: false, B: true) and (A: true, B: true)}
```

Any one of the three sets of test cases would give decision coverage.

The designation for decision coverage is *C1*. It can be quantified as the number of branches executed in a test relative to the total number of branches in the module. If there are there are no decision statements in the module, then the module is considered to have a single branch.

Condition Coverage - Condition coverage testing [MYE79] requires each condition in a decision to take on all possible outcomes at least once. Using the previous example, allowable sets of test cases, would be:

```
(1) {(A: true, B: false) and (A: false, B: true)}
    or,
(2) {(A: true, B: true) and (A: false, B: false)}
```

Decision/Condition Coverage - Decision/condition coverage testing [MYE79] requires each condition in a decision to take on all possible outcomes at least once and each decision must take on all possible outcomes at least once. The example given for decision coverage might then have one of the following sets of test cases:

```
(1) {(A: true, B: false) and (A: false, B: true) and
              (A: true, B: true)}
    or,
(2) {(A: true, B: true) and (A: false, B: false)}
```

Multiple-Condition Coverage - Multiple-condition coverage testing [MYE79] requires all possible combinations of condition outcomes in each decision to be invoked at least once. Again using the example from decision coverage, the following set of test cases would be used for multiple-condition coverage:

```
(A: true, B: false) and (A: false, B: true) and
    (A: true, B: true) and (A: false, B: false)
```

Loop Testing - Loop testing [BEI84] is a form of path testing aimed at exposing errors that typically appear within loops. The testing of loops presents special problems in that the maximum number of loop iterations can sometimes be quite large. Often, the maximum number of loop iterations is not known.

Test cases for the testing of loops are written which provide:

1. Bypassing the loop altogether

2. One pass through the loop

3. Two passes through the loop before exiting

4. A typical number of passes through the loop before exiting.

And if the maximum number of loop iterations is known, the following additional types of test cases are used:

1. One less than the maximum number of passes

2. The maximum allowable number of passes

3. Attempt one or two more than the maximum number.

1.1.2 Requirements Based Testing

Requirements based testing (or functional testing) involves the examination of system requirements and generation of test cases which will test those requirements. Some of the better known forms of requirements based testing are "equivalence partitioning", "boundary-value analysis", "cause-effect graphing", and "error guessing".

Equivalence Partitioning

Equivalence partitioning relies on the following two considerations [MYE79]:

1. Each test case should invoke as many different input conditions as possible in order to minimize the total number of test cases necessary.

2. The input domain of a program should be partitioned into a finite number of equivalences classes such that a test of a representative value of each class is equivalent to a test of any other value.

The equivalence classes are identified by taking each input condition and partioning it into two or more groups. For example, if an input condition specifies the number of values (eg, 20 through 400 dollars may be withdrawn from an account), one valid equivalence class (20 <= dollars <= 400) and two invalid equivalence classes (20 > dollars and dollars > 400) may be identified. Once the equivalence classes have been identified, test cases should be defined. The test cases should be written which cover as many equivalence classes as possible. Some overlapping of equivalence classes may be necessary in order to provide full coverage.

Boundary-Value Analysis

Boundary conditions [MYE79] are conditions directly on, above, and beneath the "edges" of input and output equivalence classes. Boundary-value analysis is similar to equivalence partioning, with the following two exceptions:

1. One or more elements should be selected from an equivalence class which can be used to test the "edge" of the equivalence class.

2. Both input and output equivalence classes are used in the selection of test cases.

Using the previous example, test cases would be written for dollars = 19, dollars = 20, dollars = 400, and dollars = 401.

Cause-Effect Graphing

Cause-effect graphing aids in selecting a high-yield set of test cases which can be used to test combinations of input [MYE79].

Using this technique, the specification describing the program is transformed into a cause-effect graph which resembles a combinatorial logic network. The "cause" is a distinct input condition or equivalence class of input conditions. The "effect" is an output condition or a system transformation. The semantics of the specification are used to link the "causes" with the "effects" into a Boolean graph format. The graph is then modified to show combinations of causes and/or effects which are impossible due to syntactic or environmental constraints. The graph is then converted to a limited entry decision table in which each column in the table represents a test case.

The basic cause effect graph symbols are shown in Figure 3.

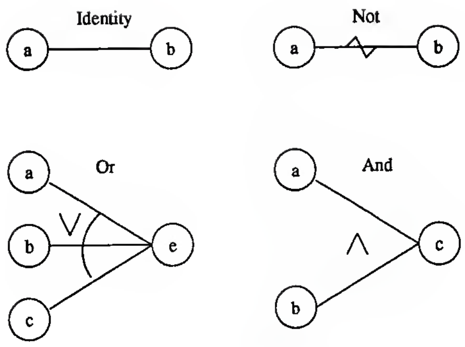

Figure 3. Basic Cause-Effect Graph Symbols

Error Guessing

Error guessing [MYE79] is largely an intuitive and ad hoc process whereby someone can think of certain probable errors and write test cases designed to test the program for those errors. For example, the value 0 is typically an error prone situation in either the input or output of a program. Test cases are written which use 0 as an input and force 0 as an output. Other examples of possible error prone situations are in non specified inputs to a program, empty or non-existent files, and interactions between two or more programs.

1.2 Regression Testing

Regression testing [MYE79, OST83, PER83, POW82] is the process of testing for possible errors which may have been inadvertently introduced into the software between versions or releases. These errors may or may not be detected by other testing methods. It is used as a means of assurance that the functions and capabilities which are in the previous, correct version of the software have not changed between versions.

"It is widely agreed that the majority of errors appearing in production software do not come from the original implementation, but rather are unintended side effects of post release program modifications." [PAN78]

Regression testing typically makes use of a subset of test cases which were used to test a previous version of software. It is generally considered uneconomical to regression test the system using the full gamut of test cases which may be available. Test cases used for regression testing should be chosen so that the most important and critical functions are exercised.

In addition to the test cases, output or detailed analysis from execution of the test cases should be maintained. This is required so that the same test cases which were run against an earlier version can be run against the current version and a comparison can be made between the two sets of outputs. Any differences which are detected between the two sets of output may require a more detailed analysis of the new version of the software.

1.3 The Cost of Software Validation

Figure 4 gives a breakdown of the costs associated with each phase in the development of five large projects [ALB81].

	Analysis and Design	Debugging	Coding and Validation
SAGE	39%	14%	47%
NTDS	30%	20%	50%
GEMINI	36%	17%	47%
SATURN V	32%	24%	44%
OS/360	33%	17%	50%

Figure 4. Comparison of Development Costs

These percentages show that more money and effort is being spent on coding and validation than analysis and design. R. W. Wolverton claims that at least 40% of software development costs are traceable to software testing costs [WOL84]. This becomes even more or a problem when the software being tested is a first release of a product. Beizer estimates this cost to be any where from 50% to 80% [BEI84].

What can be done to reduce the costs associated with software validation? An argument can be made for spending more effort in the initial design phase of a project rather than waiting to try and catch errors during the testing phases. In theory, this idea deserves much attention, but in reality, tight development schedules often force development to begin before the design has been completed. Another possible way to reduce validation costs is to simply reduce the amount of testing being performed. If software quality is to be maintained, this concept will not work. A third possible method of reducing software validation costs is to let software validate software (i.e., automate) as much as possible. Provide software tools which can be used to reduce the overhead of software validation. Let the computer perform the often tedious testing required and let the tester become a developer of test cases which can be used repeatedly to validate the software.

1.4 The Problem of Compiler Validation

Compilers are computer programs which should be validated prior to release. A program can be 100% correct, but if the compiler needed to compile the program contains errors, serious and often hard to locate bugs might occur. Because we depend so heavily on compilers to do a great amount of work for us, it is critical that they be thoroughly tested during development.

Software *bugs* or errors in a compiler can be placed into one of the following broad categories [BER83]:

1. An incorrect handling of a correct program either at compile time or at run time, or

2. A failure to detect an illegal program as illegal either at compile time or at run time.

During testing of any software program, the program should be tested for proper operation for both correct and incorrect input. In addition, the testing of compilers requires that the resulting object code be executed and tested for errors. This doubles the testing effort required for compiler validation. Thankfully, standardized compiler test cases, or test suites, have been developed to speed up the validation process.

Another problem associated with compiler validation concerns the programming language, or input, used by the compiler. Programming languages are defined in terms of their syntax and semantics. These definitions are used as specifications for the validation of the compiler associated with the language. The syntax can be formally specified by context-free grammars (eg, BNF definitions). The semantics of the language are much harder to specify than the syntax.

> "No completely satisfactory means for specifying semantics in a way that helps construct a correct compiler for the language has been found." [AHO79]

1.5 Automated Tools for Software Validation

There are two broad categories of automated tools for software validation [MIL84]. These categories are: static and dynamic analysis tools. Static analysis tools consist of static analyzers, code inspectors and standards enforcers. Dynamic analysis tools consist of coverage analyzers, test archiving systems, output comparators and test data generators. Many tools used for the purpose of software validation are combinations of both static and dynamic analysis tools.

The Software tEsting Environment Support (SEES) System [ROU85] is an example of a software validation tool which supports both static and dynamic analysis of software. It makes uses of a relational database system (ADMS) for the storage and retrieval of information critical to a large number of testing techniques.

Static analysis of programs is performed using a compiler built with the UNIX® YACC compiler-compiler [DOL80] where the grammar's rules have been augmented with write statements. As a source program is being compiled, these write statements output appropriate information into the database whenever the rule is found applicable.

Dynamic information is generated during the program's execution. This information can be generated by an interpreter or by write statements inserted in the program which enter the necessary data into the database.

The major advantage of the SEES System is that it allows someone to create his or her own tools according to the type of testing which must be performed. These tools are in the form of command files which can be used by the tester to perform database transactions and report generation after a source program has been compiled, executed and the data collected. Command files consist of utility operators, query language (SQL) commands, and UNIX system commands. Utility operators, which are written in the host language of the database management system, along with SQL commands are used to perform database transactions.

1.5.1 Coverage Analyzers

A *coverage analyzer (or execution verifier) system* [MIL81] includes a command interpreter, a run-time package, a post-execution processor, and a report generator. This type of system makes use of user specified instructions in analyzing and instrumenting the programs to be tested. As the program is executing, the instrumented software (also commonly referred to as "software probes") produces output which is collected and stored by the run-time package. The post-execution processor is responsible for producing an evaluation report of the results of the testing. The command interpreter controls the overall operation of the system. The run-time package collects and stores the results of execution. After execution, the post-execution processor analyzes data which the run time package has collected. The report generator identifies the level of testing achieved and signals when a program component was not tested.

An example of an execution verifier is the Pascal Test System (PTS) [HEL86]. This system is used for the validation of airborne safety-critical software. PTS uses a tabular module test specification, translates it into a compiled test program, and executes the test program, along with the modules to be tested, on a software emulator. The results of the test execution are collected and compared with predicted results and a report is generated. Structural coverage of the test is also reported.

The tabular module test specification which is used by PTS contains test inputs, the module being tested, the test module environment and expected test outputs. A translator is used to transform the specification into a test program. A database provides storage for the test specification so that regression testing can be performed between test specifications.

1.5.2 Test Archivers

A *test archiving system* acts as a storage and retrieval system for both test cases and results of the tests. These systems are well suited for the support of regression testing because access to test cases, output data and analysis information are important. The SEES system is an example of a test archiving system.

1.5.3 Output Comparators

Output comparators [MIL84] are used to compare two versions of a program's execution output to identify any differences which may have resulted during execution. These tools are used for both mutation analysis and regression testing. An example of an output comparator is the UNIX "diff" command [DOL80].

1.5.4 Test Data Generators

Test data generators [MIL84] are used to generate test data for the testing of specific parts of a program. This type of tool is used in structural analysis testing. Test case generators are a special form of a test data generator. These tools are used to generate test cases for the validation of compilers. Test cases are automatically generated which will randomly exercise the compiler being tested. The generation of these test cases is based on BNF extended grammars which define the language being tested.

1.6 Automated Tools for Compiler Validation

Any of the tools described in the previous section may be used to test compilers, however, there are tools which have been developed for the specific purpose of compiler validation.

Automated tools for compiler validation consist of test case generators (i.e., test data generators) and test drivers. Test case generators are capable of generating test cases which can be used to test specific parts of a compiler. Test drivers are tools which manage the execution of test cases, the archiving of test cases and results, and the reporting of testing results. The test cases which are used by test drivers can be either automatically generated or manually generated.

Some test case generators [BAZ82, CEL80, HAN70] use a formal description of the language as input to the generator. The description is capable of handling both the context-free and context-sensitive aspects of the language. The output from the generator is a set of compileable (or intentionally wrong) programs which can be used to test the lexical analyzer, the syntax analyzer, the semantic analyzer, and the error handler. These generators do not produce executable programs which can be used to test code generators and run-time systems.

Bird [BIR83] describes a PL/1 test case generator which produces random, executable, self-checking test cases. The test cases which it produces are syntactically correct and guaranteed to execute without any errors (assuming that the compiler is error free).

The test case generator described by Bird has several limitations for which test cases must be manually written. Because the generated test cases are syntactically correct, they cannot be used to test the compiler's error handler. Run-time diagnostics also cannot be tested because they cannot be made self-checking. There are still other situations which require the manual writing of test cases.

An example of a test driver which was developed specifically for the validation of UNIX and the C applications is the PERENNIAL Driver/Monitor (PDM) [PER86]. PERENNIAL also offers a validation suite for UNIX and C compiler testing which has been designed to run under control of the PDM.

The PDM is responsible for compiling each test case, linking in a support library, executing the resulting object program, and reporting the results of each test case. All output from the compilation, linking and execution phases is redirected to a file for further analysis. A nice feature of the PDM is its ability to test itself using test case files.

H. K. Seyfer [SEY82] describes the testing of the Univac UCS-Pascal compiler. A test driver, the Univac Series 1100 Test Controller System (TCS), was used to automate the testing routine. The TCS allowed the tester to select the set of test cases and the compilers, along with options and libraries to be tested. The test cases were compiled and executed. Output from the execution of the test cases was stored and compared with a predetermined test result. The TCS Status routine was used to examine the test comparison results to determine any differences in compilation, linkage or execution.

A collection of papers edited by Brian A. Wichmann and Z. J. Ciechanowicz [WIC83] discusses Pascal compiler validation and an automatic testing facility for the testing of Pascal compilers. It is a fairly simple test driver which is used primarily for producing reports of test runs. The testing facility uses comments in the test cases for reporting the nature of errors which may be detected. The major drawback to this testing facility is that there is no archival mechanism for storage of test cases and testing results.

The FORTRAN Compiler Validation System (FCVS) [HOY77] was developed by the Department of the Navy to test the conformance of elements of the FORTRAN language. The compiler is tested by compiling and executing a set of test cases. If the compiler fails to compile the set of test cases, the test case causing the failure is eliminated. The set of test cases is then recompiled and executed. Test results produced by the execution of each test case indicate whether the execution passed or failed.

These and other automated tools for software testing tend to be very specialized items. They are designed to test a single language, or in some cases, a specific system in which the language is used. They often lack the flexibility required by the testing organization to be modified in order to suit changing needs.

The Automated Regression Test System (ARTS), which is presented in the following sections, is a test driver which may be used for the regression testing of compilers and interpreters. Unlike other test drivers, it is not limited to a single compiler or interpreter. This flexibility allows the ARTS to be tailored to a particular project and the project's own requirements.

CHAPTER 2

REQUIREMENTS AND CAPABILITIES FOR THE AUTOMATED REGRESSION TEST SYSTEM (ARTS)

The previous section presented various software validation techniques and tools which may be used in support of software testing. This section describes the requirements and capabilities necessary for the Automated Regression Test System (ARTS), which can be used for regression testing both compilers and interpreters.

Figure 5 presents an overview of a typical testing session using the ARTS. The figure represents both compiler and interpreter validation sequences. The following example of a typical testing session applies to the interpreter case shown in Figure 5.

As an example, assume that Product XYZ depends on a Prolog interpreter. If a new version of the Prolog interpreter is required, due to performance considerations or general enhancements, both functional tests of the modifications and regression tests of the parts of the interpreter which were not changed must be made prior to installation into the product. In order to run the regression tests, a person using any standard CRT terminal can enter "arts" from the UNIX shell and immediately get a menu of items from which to select. He or she would next select the item which provides for the selection of test cases to be executed. The main menu is cleared and a secondary window or form is created which prompts for the language to be used (in this case "Prolog"), the name of the existing compiler or interpreter, the name of the compiler or interpreter to be tested and the names of the classes of test cases to be executed. After making these selections, the ARTS user would then select an option to return to the main menu. Execution is begun by selecting the test run execution option. The ARTS would then instruct the existing Prolog interpreter to execute the test cases selected and save the results in an output file.

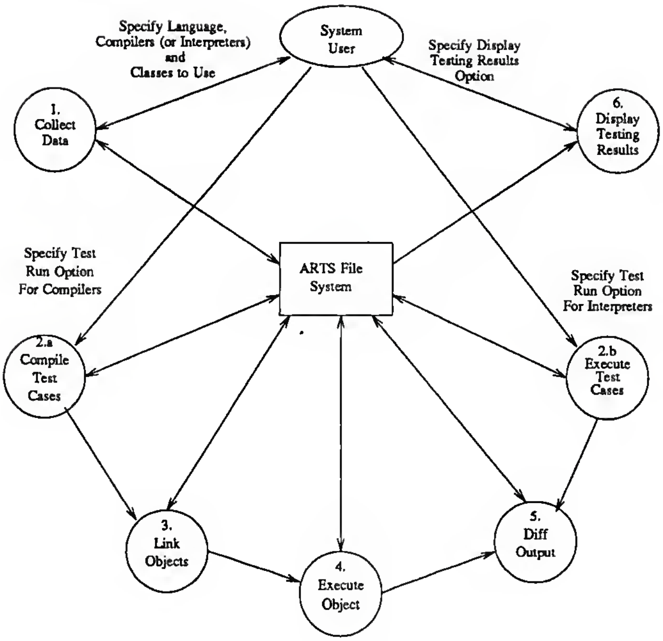
Specify Language, Compilers (or Interpreters) and Classes to Use
System User
Specify Display Testing Results Option
1. Collect Data
6. Display Testing Results
Specify Test Run Option For Compilers
ARTS File System
Specify Test Run Option For Interpreters
2.a Compile Test Cases
2.b Execute Test Cases
3. Link Objects
4. Execute Object
5. Diff Output

Figure 5. Testing Session Example

Next, the interpreter under test would be instructed to execute the same test cases and save the results in another output file. Once all test cases have been executed by both interpreters, the resulting outputs are compared by the ARTS for any differences. The testing results, both the results of the comparison and an indication as to whether or not any differences were detected, are stored in the ARTS file system for later examination. An analysis of the testing results is sent to the user.

Several requirements apply to software development tools in general. The tool must do what it was designed to do, without errors, in an efficient and cost effective way. The user interface needs to be well planned so that someone using the tool needs little training in its operation. The tool should also be flexible so that it can be used by a wide population of users.

> "Regression tests should be easy to run, should not involve extensive manpower or set-up time, and results should be easy to analyze. If these tests violate any of these criteria, they probably won't be run with enough frequency to be of any value." [BOW83]

The following paragraphs in this section present requirements and capabilities which are necessary for the Automated Regression Test System (ARTS). Its general requirements include:

1. Storage and retrieval of test cases
2. Compilation and/or execution of test cases
3. Storage of the results of execution
4. Analysis of the results.

The ARTS must be a flexible, easy to use automated test driver which allows the user to perform regression testing for compilers and interpreters. It needs to be a flexible system so that any compiler or interpreter which runs on a UNIX operating system can be regression tested.

A hierarchical file system should be used for the archiving of test cases, input data files, output data files, information for compiling and running the test cases, and results of the testing. The file system must be designed to handle any number of languages, compilers, and test cases (there is, however, an operating system limit on the number of files which can be created). The file system should be well structured so that it is free to grow as new test cases are added to the ARTS.

The user should be allowed to retrieve information about the structure and contents of the file system whenever he or she is in the process of installing test cases. The requested information needs to be presented to the user as a listing of languages supported, compilers/interpreters supported, and classes of test cases currently installed.

Test cases to be compiled and/or executed are selected on a per class basis. Each class consists of a single file containing one or more test cases. The user should be allowed to select any number of classes or all classes associated with the language to be used for the regression test. For example, several classes may exist for the Prolog language. These classes include i/o operations, looping operations, assignment operations, and other similar classes of test cases. A person using the ARTS may wish to choose all classes of test cases or a set of the classes.

Because of the differences of compiling, linking, and executing programs between languages, the user must specify completely how the tests are to be executed. These specifications need to be collected and stored in a file which can be later used as a command or script file by the UNIX shell. The concept of execution specification script files give the ARTS the flexibility needed to support the regression testing of any compiler or interpreter running on the host UNIX system. Examples of typical execution specifications are shown in Figure 6.

Several things should be noted in the sample execution specifications. There are two key words which appear in each execution specification. These words are: $COMPILER and $INTERPRETER. One of the key words must always appear in the execution specification. The key word is to be replaced by a path (e.g., /bin/cc) to the compiler or interpreter to be tested prior to the beginning of the test run. The file "test_cases" (with or without a suffix) will always be used to store the test cases for each class. Compiler or interpreter options may be included on the command lines.

The execution specification script file should be built whenever the ARTS user installs a new test case file and there is no associated execution specification. There should also be an option

```
Example 1:
      $COMPILER test_cases.c
      a.out
Example 2:
      $COMPILER -I/tom/include test_cases.c
      a.out
Example 3:
      $COMPILER -Dtom -O -o ctest test_cases.c /tom/lib/lib.a
      ctest
Example 4:
      $INTERPRETER <<!
      ['test_cases.pl'].
      !
Example 5:
      $INTERPRETER test_cases
```

Figure 6. Typical Execution Specifications

which will allow someone to build a new execution specification file on demand. The interface to this function should be a simple line oriented editor which will allow up to 20 lines to be entered.

Once the execution specification has been built, it should be tested to verify that it is syntactically correct. Verification that the user has entered either the $COMPILER or $INTERPRETER variables and the word "test_cases" should be made. If an error is detected, an error message should be displayed informing the user of the error. Prior to installing the execution specification into the ARTS file system the $COMPILER or $INTERPRETER variable should be replaced with the path to the appropriate compiler or interpreter and the appropriate suffix added to the word "test_cases".

If a compiler is being tested, the selected set of test cases must be compiled, on a per class basis, by each compiler. If the compiler fails to compile the class of test cases, the compiler error message should be saved for analysis purposes and no execution should be performed for that class. Execution should continue with the next class in the sequence of classes selected.

The same situation holds true for the link editor. If the link editing phase fails, the error message should be saved for analysis purposes and no execution should be performed for that class.

Analysis of testing results consists of comparing the output from a test run of each class with that of another. If any differences are detected between the collected data, then the compiler or interpreter is said to have failed the testing for that class. Analysis of these results (either "pass" or "fail") and the resulting output from the comparison must be stored in a file associated with the class being tested.

A log file should be kept for each class of test case. Information to be stored in this log file consists of the name of the person installing a test case file, the date and time of installation, the name of the person performing a test run, the date and time of the test run, the compilers being used for the test and the pass/fail status.

After a test run, an ARTS user should receive a file containing the date and time of the test run, the language used, the compilers tested, the class or classes selected, the pass/fail status of the test run, and a listing of the differences detected. This file can be placed in the user's rje directory.

A simple report generation function must be provided. The report should be allowed to be generated after each test run has been performed and an analysis of the results has been made.

Test cases may be deleted on a per class basis (i.e., all test cases for a given class are deleted). All data files for the associated class must also be deleted from the file system.

On-line help should be available to the user. A user definable help file will be used to store any help information which an ARTS administrator may feel to be important to the ARTS user. Typical information to be stored in this file might include information on how to build an execution specification or information on how to structure test cases.

Limitations associated with the ARTS include the following:

1. The ARTS should be implemented as a single user system. No file locking mechanisms or other protection schemes are required.

2. It does not validate compilers and/or interpreters between two machines or operating systems.

3. There are no requirements for batch operation (i.e., test execution cannot be scheduled).

4. Test cases must be written in the proper format so that analysis of testing results can be produced.

5. Test runs cannot be restarted if a system failure occurs (e.g., a power failure). The user must make a new execution request.

6. There are certain features which cannot be automatically tested. For example, input from a terminal.

CHAPTER 3

DESIGN DESCRIPTION FOR THE AUTOMATED REGRESSION TEST SYSTEM

Chapter 2 presented the overall requirements for the ARTS and gave a typical testing session example which involved regression testing a Prolog interpreter. This chapter presents the detailed design derived from those requirements and expands on the previous testing session example to describe the actual data flow (refer to Figure 7) and control flow. The functional specification for the system is shown in Appendix A. The source code for the ARTS is presented in Appendix B.

Using the example from the previous chapter, the person using the ARTS was presented a menu of functions from which to make a selection. Once the option to select test cases for a test run was chosen, the main menu was erased and a form was drawn on the screen. All screen i/o associated with menu selection, form entry and display, error messages and status messages is handled by the *User Interface* subsystem. The language "Prolog" was entered as the language to be used for testing. The ARTS file system is checked for the Prolog language. If the Prolog language is not installed, the User Interface displays the appropriate error message. All ARTS file system operations are handled by the *File System Manager* subsystem. File system operations include the installation, deletion, selection and retrieval of system data.

After all the selection data has been entered by the ARTS user, the File System Manager builds a file which contains paths to all test case files to be executed. If a compiler is being tested, the file would contain paths to all test case files to be compiled. Next, two script files are built for each class of test case selected. These script files (shown as Exec_spec1 and Exec_spec2 in Figure 7) contain the contents of the exec_spec file (built when the test cases were installed) with $INTERPRETER variable replaced by the path to the appropriate Prolog interpreter. All

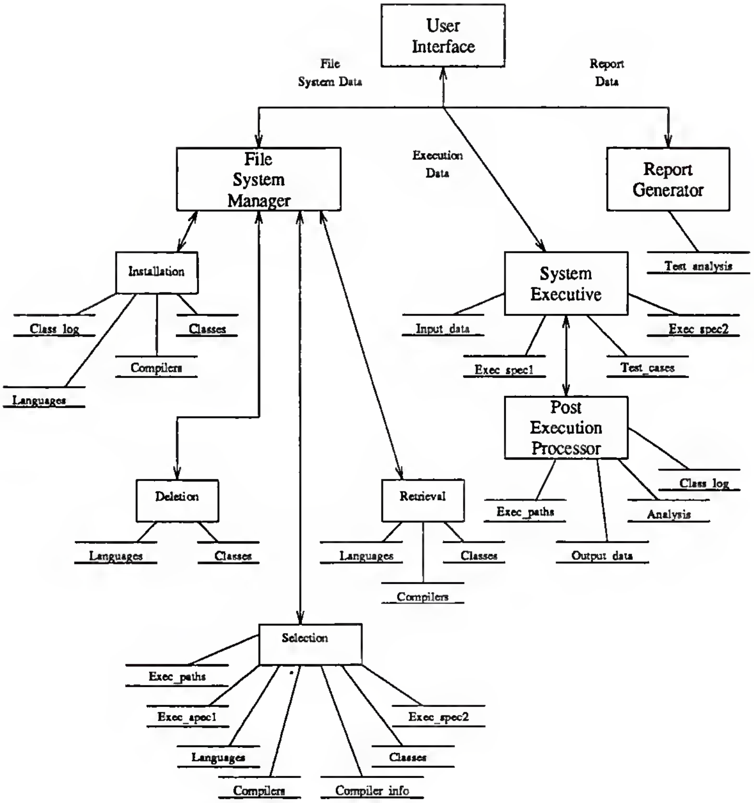

Figure 7. ARTS Hierarchical Data Flow Diagram

Exec_spec1 and Exec_spec2 files are built in the same directory as the one which contains the test_cases file. Control is passed back to the User Interface after all Exec_spec files have been built. The ARTS user now selects the option to begin the test run. Control is passed to the *System*

Executive subsystem. The System Executive builds shell command lines which will execute the Exec_spec1 and Exec_spec2 files and save the output in separate Output_data files. After all test cases have been executed, control is passed to the *Post Execution Processor* subsystem. The Post Execution Processor uses the Exec_paths file to build shell command lines to make a comparison (via the UNIX "diff" command) of each Output_data file. The results are placed in the Analysis file. Overall testing results for all classes of test cases executed are sent to the ARTS user. Control is passed back to the User Interface where the person using the ARTS can select to generate a report using the *Report Generator* subsystem or select another option from the main menu.

2.1 The User Interface

The User Interface relies on the *CURSES* [ARNOLD] package for the handling of the terminal screen i/o functions. *CURSES* is a collection of functions which present a high level screen model to the programmer, while dealing with issues such as terminal differences and optimization of output.

The terminal screen is divided into three regions or sections. The first section (normally, lines 1-22) is for displaying the system menus, entry forms, and file system data. The second section (normally, line 23) is the status message line. The third section, which will always be the last line on the screen, is the error message line. If the terminal is capable of handling inverse video, all error and status messages will be displayed in this mode.

The *main menu* provides a selection to a series of forms (i.e., installation, deletion, retrieval, and selection), or to either a "execute", "report", "display", "help" or "exit" function. The "execute" function provides entry into the System Executive for startup of the test run. The "report" function provides a hardcopy printout of the results of a test run. The "display" function displays the results of a test run on the user's terminal. The "help" function is used for displaying on-line help information the testing results. All help information is stored in a file which can be easily edited to suit the needs of the user. The testing results are stored in a file in the user's rje directory. "Exit" returns the terminal to its "pre-curses" state and exits the ARTS.

Selection of a menu option is achieved by moving the cursor (via the TAB key) to the desired option and typing an "x". Only the TAB and "x" keys are recognized by the menu. Any other key (with the exception of BREAK and DEL) which may be typed is ignored. The cursor will always be positioned at the first option upon entry into the menu.

Data entry into a field on a form can be made by entering the data and pressing the RETURN key. A 'q' may be entered to cause a return to the main menu. A '?' can be entered into many of the fields to get a display of the contents of the file system for the named object. This display appears on a new window which is overlaid on top of the existing form. All data is verifyed immediately after it has been entered in a field. If an error is detected, an error message will be written, the field will be cleared and the cursor will be positioned at the beginning of the field.

Typing a BREAK or DEL while in either a menu or a form will cause the terminal to be returned to the "pre-CURSES" state and an exit will be taken from the ARTS.

The *installation* form allows the user to install languages, compilers, test case classes, execution specifications, and test cases in the file system. When installing test cases, the user is required to enter the language, the compiler, the class, the file containing the source test cases, the execution specification, the input data file (if any), and the suffix which the test case file must have (if any). The execution specification is entered on a separate window from the *installation* form. All fields will be cleared, and a status message written, after all the information has been successfully stored by the File System Manager.

The *deletion* form allows the user to delete a class of test cases. The user will be required to enter the language and the class. All data files, including the test case file and the execution specification, and directories will be deleted from the file system for that class. All fields will be cleared, and a status message written, after the class has been deleted by the File System Manager.

The *retrieval* form allows the user to retrieve a class of test cases. The user will be required to enter the language, compiler, and the class. All data files, including the test case file, will be retrieved from the file system for that class. The retrieval process constructs a copy of the files; therefore, the contents of the file system are left undisturbed. All fields will be cleared, and a status message written, after all the information has been retrieved by the File System Manager.

The *selection* form provides for the selection of compilers and classes of test cases which will be used for a test run. The user will be required to enter the name of the language, the names of the two compilers, the UNIX paths to the two compilers (if they have not been previously installed into the ARTS file system), and the names of the classes of test cases to be used. The word "all" can be entered as a keyword to specify all classes are to be used for the language selected. Note that two compilers are required, one is the standard and the other is the compiler being tested. The selection form contains a scrolling region which allows the user to select any number of classes of test cases. A "." on a line by itself is used to signal the end of the selection of classes. Any number of classes of test cases can be selected. Control is then passed to the File System Manager.

2.2 The File System Manager

The File System Manager is responsible for all file system activity. File system activities include installation, deletion, and retrieval of test cases and data files. It also maintains the structural integrity of the file system.

2.2.1 File System Structure

The file system is a hierarchical structured file system which uses the UNIX directory structure for file storage. The root of the file system is the directory "arts". The ARTS variable should be set to the path of the root directory. For example:

```
ARTS=/usr/arts ; export ARTS
```

A diagram of the file system structure is shown in Figure 8 (directories are in bold and temporary files are shown in italics).

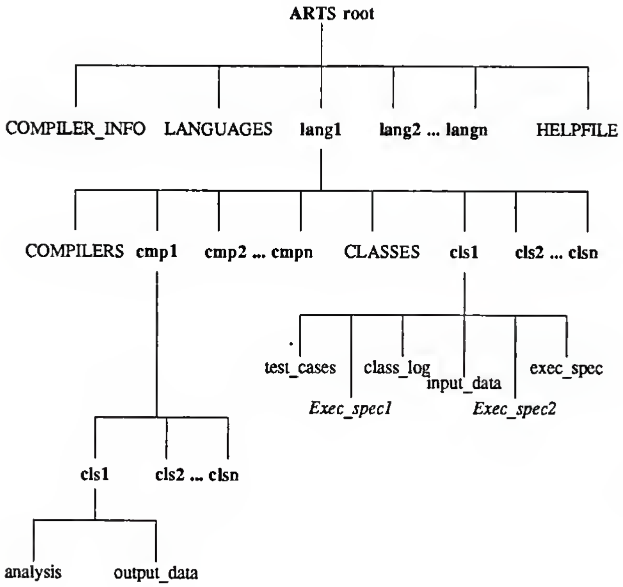

Figure 8. File System Structure

Three ASCII files are used by the File System Manager for tracking the structural makeup of the file system. These are the "LANGUAGES" file, the "COMPILERS" file, and the "CLASSES" file. The LANGUAGES file contains the languages installed in the file system. The COMPILERS file contains the compilers installed in the file system for a given language. The

CLASSES file contains the classes of test cases in the file system for a given compiler.

Directory names, which are derived from the name of the language, compiler and class, are limited to 14 characters. The algorithm for doing this is presented later in this section.

Directories for each class appear as subdirectories to a language and to each compiler for that language. The class directories, which are subdirectories to a language, contain the following files: test_cases, input_data, class_log, exec_spec, Exec_spec1 and Exec_spec2. The class directories which are subdirectories to a compiler contain the analysis file and the output_data file.

On-line help information is placed into the HELPFILE. This information gives a brief description of the main menu and each of the forms.

Each line in the COMPILER_INFO file is divided into three fields. These are: the compiler name, the system path to the compiler, and any suffix (eg, .c) which may be required by the compiler. Each field is separated by a colon.

The test_cases file contains all test cases associated with the class of testing. The suffix (if any) which this file must have is derived from the COMPILER_INFO file.

The input_data file contains all input data which may be used by the test cases. This is typically input data which may be needed to test I/O operations.

The exec_spec file is the primary execution specification file. This is a script file which contains the commands necessary for the system to compile, link, and/or execute the test_cases file. One and only one of the keywords *$COMPILER* or *$INTERPRETER* must appear in the exec_spec file. Also the word "test_cases" with the appropriate suffix must appear. Examples of exec_spec files were shown in the previous section.

The Exec_spec1 and Exec_spec2 files are both temporary command script files which are built by ARTS during the execution of a test run. They contain a copy of the exec_spec file with the appropriate path to the compiler or interpreter under test substituted for the $COMPILER or $INTERPRETER keyword.

The class_log file contains the name of the person who installed the test_case file and the date of installation. These entries are appended to the file so that a record of installation activity is maintained.

The output_data file contains the output results from the compilation, linkage, and/or execution of the test cases.

The analysis file contains the analysis of the testing results for the associated class of testing. This will be either a "PASS" or "FAIL". Also, the comparison results, the id of the person requesting the test run, and the date are stored.

2.2.2 Installation

Installation procedures are provided for the installation of test cases and the installation of executions specifications. Both of these options are provided on the main menu.

Installation of Test Cases

When installing test cases, parameters required by the File System Manager are: root directory of the ARTS file system, language, compiler, class, name of test case file, name of input data file (if any), and the execution specification.

If the compiler is not included in the COMPILER_INFO file, the user will be required to enter the path to the compiler and also any suffix required by the test_cases file. This information, along with the name of the compiler, is then appended to the COMPILER_INFO file. The File System Manager will check the path to the compiler for accuracy. If the compiler, or interpreter, is not found, an error message will be displayed.

If the root directory, language directory, compiler directory, or class directory does not exist, the appropriate new directory is built. The class directory which is a subdirectory to a compiler is not built until test cases have been compiled and executed for that compiler. Directory names, other than the root directory, are built from object (language, compiler, or class) names. The first fourteen characters in the object name are used. Any special character (eg, * or ?), characters which may cause conflicts (eg, @ or /) and all blanks are transformed to an underscore. All upper case alpha characters are converted to lower case. If a language, compiler, or class is entered which may cause conflicts with an existing directory for another class, an error message is given to the user that a conflict exists and another name should be used. This conflict occurs when the first fourteen characters of the object are the same as the first fourteen characters of another object.

If a LANGUAGES file, COMPILERS file or CLASSES file does not exist, the appropriate file is created and the name of the object being installed is written to the file; otherwise, the name of the object is appended to the associated file.

The test_cases file is built by copying the contents of a user specified test case file into the test_cases file. If a test_cases file currently exists, its contents are overwritten. Test case files are built using test_cases as the file name with any suffix required appended to it. For example, the Prolog interpreter requires that file names in a ".pl"; therefore, the name of the file would be test_cases.pl.

The input_data file is built by copying the contents of a user specified data file into the input_data file. If an input_data file currently exists, its contents are overwritten.

The exec_spec file is created from user supplied information. The user is prompted for the information after all test data files have been built. The user input is checked for the keywords:

$COMPILER or *$INTERPRETER*. If neither of these keywords are detected. An error message is written, the screen is cleared and the user is again prompted for the execution specification. The word "test_cases*suffix*" must also be entered by the user. The user may quit by entering a "q" on the first line and pressing RETURN.

The name of the user performing the installation and the current date are retrieved from the UNIX system and placed in the class_log file. If a class_log file currently exists, the new information is appended to the class_log file.

If an analysis file or output_data file exists, they will be removed.

Installation of Execution Specification

This function allows a user to change any existing exec_spec file in the file system without disturbing existing test cases.

Installation of an execution specification requires the root directory of the file system, language, compiler, class and the execution specification. An error message is returned to the user if the file system, language, compiler, or class does not exist.

The building of the execution specification is the same as that described for the installation of test cases. Any existing exec_spec file is overwritten by the new exec_spec file.

2.2.3 Selection

The *Selection* form is used for the selection of all parameters needed in the execution of the test run. Required parameters for the selection of compilers and classes of test cases are: the language, two compilers or interpreters, the system path to the compilers or interpreters, and one or more classes. All parameters are checked for installation in the ARTS file system. If the parameter has not been installed, an error message is displayed and the user is again prompted for the parameter. The COMPILER_INFO file is checked for the name of the entered compiler or interpreter, and if found, the path is displayed.

The *$COMPILER* or *$INTERPRETER* variables in the exec_spec must be translated into the UNIX system path to the applicable compiler or interpreter. This is done by first copying the exec_spec file into an Exec_spec1 and Exec_spec2 file. The variable in the Exec_spec1 file is then replaced by the path parameter to the compiler or interpreter being tested. The Exec_spec2 file's variable is replaced by the path parameter to the compiler or interpreter standard. Both Exec_spec1 and Exec_spec2 files are removed immediately after use. Note that for each class which is selected for a test run, there is one Exec_spec1 and one Exec_spec2 file.

The exec_paths file is used to store all UNIX paths needed by both the System Executive and the Post-Execution processor. Each line of the file contains four fields, separated by a ":". Field 1 is the name of the class being tested. Field 2 is the path to the directory containing the exec_spec file. Field 3 is the path to the compiler directory for the compiler under test. Field 4 is the path to the compiler directory for the compiler standard. The exec_paths file is built by the File System Manager from parameters passed to it from the User Interface.

After all Exec_spec1 and Exec_spec2 files and the exec_paths file have been built, control is passed back to the User Interface.

2.2.4 Deletion

Deletion parameters are passed from the User Interface to the File System Manager. Required parameters are the language and the class to be deleted. All parameters are checked for installation in the ARTS file system. If the parameter has not been installed, an error message is displayed and the user is again prompted for the parameter.

All data files for the class to be deleted are removed from the file system. The subdirectories for that class are also removed. The class entry is removed from the classes file. If an error occurs, an error message is printed and control is passed to the deletion form. If there are no errors, a status message is printed which states that the class has been deleted from the file system.

2.2.5 Retrieval

Retrieval parameters are passed from the User Interface to the File System Manager. Required parameters are the language, the compiler, and the class. All parameters are checked for installation in the ARTS file system. If the parameter has not been installed, an error message is displayed and the user is again prompted for the parameter.

All data files (i.e., test_case, input_data, class_log, analysis, exec_spec, and output_data) are retrieved for the specified class. The files are copied into the user's rje directory.

2.3 The System Executive

The System Executive controls the compilation and execution of the test cases which have been selected for testing. Control is passed to the System Executive from the User Interface. No parameters are required from the User Interface.

Commands necessary for the execution of the Exec_spec1 and Exec_spec2 scripts and redirection of "stdout" and "stderr" to the proper output_data file are built using the exec_paths file. An example of these commands is shown below.

```
sh Exec_spec1 1> $ARTS/c/c5.0/i_o/output_data 2>&1
sh Exec_spec2 1> $ARTS/c/c4.0/i_o/output_data 2>&1
```

In this example, the C language is being tested using a "c4.0" compiler as the standard and a "c5.0" compiler as the compiler under test. Both "stdout" and "stderr" are redirected to the output_data file for the associated compiler. Status messages are sent to the user at the beginning of each execution of the Exec_spec file. The Exec_spec file is removed after its execution.

A "chdir" is made to the directory in which the test_cases file is located prior to the execution of the Exec_spec file. This gives the user the ability to read input data from the input_data file, if one is present. If the directory which is to contain the output file is not installed, a appropriate directory is created.

Control is passed to the Post-Execution Processor from the System Executive after all Exec_spec files have beed executed and removed.

2.4 The Post-Execution Processor

The Post-Execution Processor is responsible for performing a UNIX "diff" operation on the output_data files and analyzing the results of the testing.

Commands necessary for executing the diff and redirecting the output to the associated analysis file are built using the exec_paths file. An example of these commands is shown below.

```
diff $ARTS/c/c5.0/i_o/output_data $ARTS/c/c4.0/i_o/output_data >
       $ARTS/c/new/i_o/analysis
```

This example shows the output from the "c5.0" compiler (the compiler under test) being "diffed" with the output from the "c4.0" compiler. The output from the resulting diff execution is always placed in the analysis file of the compiler under test.

After each diff operation, the resulıs are tested to see if any differences have been detected. If there are no differences, the current date and the word "PASSED" are appended to the class_log file for the compiler under test. If there are differences, the current date and the word "FAILED" are appended to the class_log file.

The "test_analysis" file is built by the Post-Execution Processor for reporting purposes. This file contains the id of the user performing the test run, the date, the language, the compilers and the classes of test cases being tested. The words "PASSED" or "FAILED" are appended to the file, along with the diff output, for each class being tested. This file is sent to the user's rje directory after all output_data files have been analyzed. Control is then passed to the System Executive.

2.5 The Report Generator

The Report Generator is responsible for producing a hardcopy printout of the testing results. The "test_analysis" file is sent to the local printer. The command necessary for sending the analysis file to the printer may need to be modified to suit the host environment.

CHAPTER 4

FORMAT OF A TEST SUITE

Several papers have been written specifically dealing with the design of test cases for compiler testing [SEY82, WIC76, BER83, WIC83]. The authors present varying methodologies on the subject, each with its own advantages and disadvantages. The purpose of this section is not to present a design methodology, but rather to suggest some things to be aware of when designing test cases to be used with the ARTS.

Several levels of correctness can be tested for in compiler validation using well designed test cases. These are [GRA80]:

- Lexical correctness - are the program tokens well formed?
- Syntactical (or context-free) correctness - is the program structure derived according to its formal context-free description?
- Compile-time correctness: can the program be compiled without diagnostic messages?
- Run-time correctness - can the compiled program be executed without run-time errors such as infinite loops or arithmetic faults?
- Logical correctness - can the program be executed and produce the expected results?

In the case where an interpreter is being tested, all levels of correctness apply with the exception of compile-time correctness.

Through regression testing, the ARTS can test for each level of correctness.

Output data to be used in the analysis of the testing results comes directly from the compiler (or interpreter), link editor, run-time system, and/or test cases. Statements which write data to "stdout" or "stderr" should be embedded in the test cases.

Any number of test cases may be used in the test_case file; however, they should all test the same class of feature.

There are several constraints to be aware of when designing test cases to run under the control of the ARTS.

1. All input files are named "input_data" and are located in the same directory as the source file. Only one input file can be associated with each "test_cases" file.

2. No input from a terminal is permitted.

3. Because the ARTS is running under the control of the UNIX operating system, test cases should not be written which might affect its operation. For example, if the UNIX shell is being test, the statement "rm ./*" might have disastrous results.

4. Infinite loops should be avoided.

5. Any temporary files which are created should be removed upon exit from the program.

A sample portion of a suite of test cases used to test Prolog interpreters is shown in Appendix C.

CHAPTER 5
IMPLEMENTATION

The ARTS has been compiled and tested on a DEC VAX 11/780 running 4.3 BSD UNIX, an AT&T 3B15 running UNIX System V R2.1, and an Amdahl running UTS 5.2u370.

The *Curses* terminal handling package is used for all screen i/o operations. There are different versions of the Curses package. The ARTS has been designed to compile and run with any of the known versions of Curses.

A makefile is provided for building the ARTS object module. System administrators are free to set up local system libraries to meet their specific needs. Certain libraries which may or may not have to included in the makefile are "curses", "termcap" and "termlib". It may sometimes be necessary to contact the local system administrator for help in setting up the compile line parameters.

The report.c program module builds a UNIX shell command line for sending test run analysis to a printer. This program may have to be modified slightly to agree with the local environment.

There are approximately 2300 non-commentary lines of C source code. The object module contains 220489 bytes.

CHAPTER 6

CONCLUSIONS/EXTENSIONS

Validation of software is both a tedious and labor intensive undertaking. Every effort should be made to improve this task in order to lower the overall cost of the software, while at the same time raising its quality.

Many automated tools have been developed to aid the tester in the validation of software. The majority of these tools, however, are "tied" to the software or system being tested. They do not provide enough flexibility to be used with other systems or languages. The tools often become as complex as the software being tested.

The ARTS was developed as a tool for the regression testing of compilers and interpreters. It is flexible enough to be used with any compiler or interpreter which can run under the UNIX operating system. A minimum amount of training is required to train a novice user in the operation of the system.

Several possible extensions could be added. These include:

1. editing individual test cases
2. stop/start of test run
3. testing compilers between languages,
4. batch processing of test runs

The only method for editing individual test cases is for the user to retrieve the test case, edit it, and install it back into the file system. An option could be added to allow the user to choose a test case file to be edited without having to remove it from the file system.

If a test run is stopped by a BREAK or DEL, execution is halted, any cleanup operations are performed and the ARTS is exited. Rather than exiting, control could be returned to the main menu and an option for restarting the test run could be selected.

The ARTS is limited to testing compiler operation for a given language. The program could be modified so that validating compiler operation between languages was possible. For example, assume that the i/o operations associated with both a C compiler and a Pascal compiler needed to be tested. Two suites of test cases, one in C and the other in Pascal, would be executed. Both suites of test cases are functionally the same. The results of the execution could be compared for any observable differences between languages.

The ARTS system is strictly an interactive system. The user selects the set of test cases to run, starts the execution and waits for the results. This presents a problem if the system is heavily loaded or a long test run is to be made. Batch processing would allow the user to select the test cases, select a starting time for the tests to be run in the background and retrieve the test results after execution. The user would then be free to do other work without having to wait for the execution to complete.

REFERENCES

[AHO79] Alfred V. Aho and Jeffrey D. Ullman, *Principles of Compiler Design*, Reading, Mass.: Addison-Wesley Publishing,1979.

[ALB81] David S. Alberts, "The Economics of Software Quality Assurance", *Tutorial: Software Testing & Validation Techniques*, edited by E. Miller and W. E. Howden, IEEE Computer Society Press, IEEE Catalog No. EHO 180-0, 1981.

[ARNOLD] Kenneth C. R. C. Arnold, "Screen Updating and Cursor Movement Optimization: A Library Package", Department of Electrical Engineering and Computer Science, Univ. of California, Berkeley, California.

[BAZ82] Franco Bazzichi and Ippolito Spadafora, "An Automatic Generator for Compiler Testing", *IEEE Transactions on Software Engineering*, Voume SE-8, No. 4, pp 343-353, July 1982.

[BEI84] Boris Beizer, *Software System Testing and Quality Assurance*, New York: Van Nostrand Reinhold, 1984.

[BER83] D. M. Berry, "A New Methodology for Generating Test Cases for a Programming Language Compiler", *ACM Sigplan Notices*, Volume 18, No. 2, February 1983.

[BIR83] D. L. Bird and C. U. Munoz, "Automatic Generation of Random Self-Checking Test Cases", *IBM Systems Journal*, Vol 22, No 3, pp 229-244, 1983.

[BOE84] B. W. Boehm, "Software Life Cycle Factors", *Handbook of Software Engineering*, edited by C. R. Vick and C. V. Ramamoorthy, New York: Van Nostrand Reinhold, 1984.

[BOW83] J. B. Bowen and M. F. Moon, "Experience in Testing Large Embedded Software Systems", *National Conference on Software Test and Evaluation*, National Security Industrial Association, February 1983.

[CEL80] A. Celentano, S. Reghizzi, P. Vigna and C. Ghezzi, "Compiler Testing using a Sentence Generator", *Software-Practice and Experience*, Volume 10, pp 897-918, 1980.

[DOL80] T. A. Dolotta, S. B. Olsson and A. G. Petruccelli, editors, *UNIX User's Manual Release 3.0*, Murray Hill, N.J.: Bell Telephone Laboratories, 1980.

[FAI78] R. E. Fairley, "Tutorial: Static Analysis and Dynamic Testing of Computer Software", *Computer*, Volume 11, No. 4, April 1978.

[HAN70] K. V. Hanford, "Automatic Generation of Test Cases", *IBM Systems Journal*, Volume 9, No. 4, 1970.

[HEL86] K. A. Helps, "Some Verification Tools and Methods for Airborne Safety-Critical Software", *IEE Software Engineering Journal*, Volume 1, No. 6, November 1986.

[HOW81] W. E. Howden, "A Survey of Dynamic Analysis Methods", *Tutorial: Software Testing & Validation Techniques*, edited by E. Miller and W. E. Howden, IEEE

Computer Society Press, IEEE Catalog No. EHO 180-0, 1981.

[HOY77] P. M. Hoyt, *The Navy FORTRAN Validation System*, ADPE Selection Office, Department of the Navy, Washington, DC. (USGR Ref No. AD A039 770), May 1977.

[MIL84] E. F. Miller, "Software Testing Technology: An Overview", *Handbook of Software Engineering*, edited by C. R. Vick and C. V. Ramamoorthy, New York: Van Nostrand Reinhold, 1984.

[MIL81] E. Miller, "Introduction to Software Testing Technology", *Tutorial: Software Testing & Validation Techniques*, edited by E. Miller and W. E Howden, IEEE Computer Society Press, IEEE Catalog No. EHO 180-0, 1981.

[MYE79] G. J. Myers, *The Art of Software Testing*, New York: John Wiley & Sons, 1979.

[OST83] Leon Osterweil, "Status and Directions for Software Testing and Evaluation Tools", *National Conference on Software Test and Evaluation*, National Security Industrial Association, February 1983.

[PAN78] D. J. Panzl, "Automatic Software Test Drivers", *Computer*, Volume 11, No. 4, April 1978.

[PER83] W. E. Perry, *A Structured Approach to Systems Testing*, Wellesley, MA: QED Information Sciences, p 87, 1983.

[PER86] "PERENNIAL Driver/Monitor Maunual", PERENNIAL Corp., Santa Clara, CA, 1986.

[POW82] P. B. Powell, *Software Validation, Verification, and Testing Technique and Tool Reference Guide*, NBS Special Publication 500-93, U. S. Dept of Commerce, September 1982.

[ROU85] N. Roussopoulos and R. T. Yeh, "SEES-A Software Testing Environment Support System", *IEEE Transactions on Software Engineering*, Vol SE-11, No 4, pp 355-366, April 1985.

[SAI84] S. H. Saib, "Formal Verification", *Handbook of Software Engineering*, edited by C. R. Vick and C. V. Ramamoorthy, New York: Van Nostrand Reinhold, 1984.

[SEY82] H. K. Seyfer, "Tailoring Testing to a Specific Compiler - Experiences", *ACM Sigplan Notices*, Volume 17, No. 6, June 1982.

[STU81] L. G. Stucki, "New Directions in Automated Tools for Improving Software Quality", *Tutorial: Software Testing & Validation Techniques*, edited by E. Miller and W. E. Howden, IEEE Computer Society Press, IEEE Catalog No. EHO 180-0, 1981.

[VOG85] U. Voges and J. R. Taylor, "Systematic Testing", *Verification and Validation of Real-Time Software*, edited by W. J. Quirk, Berlin: Springer-Verlag, 1985.

[WIC76] B. A. Wichmann and B. Jones, "Testing ALGOL 60 Compilers", *Software Practice and Experience*, Volume 6, 1976.

[WIC83] B. A. Wichmann and Z. J. Ciechanowicz, "Developing and Testing Procedures", *Pascal Compiler Validation*, edited by B. A. Wichmann and Z. J. Ciechanowicz, Chichester, England: John Wiley and Sons, 1983.

[WOL84] R. W. Wolverton, "Software Costing", *Handbook of Software Engineering*, edited by C. R. Vick and C. V. Ramamoorthy, New York: Van Nostrand Reinhold, 1984.

APPENDIX A

Functional Specification

Function INSTALL(test_cases)

```
SUBACTION: prompt user for language.
  INPUTS:
    OBJECT: language
      TYPE: character string (keyboard)
      CONSTRAINTS: <=78 ascii characters
                   terminated by RETURN
    OBJECT: LANGUAGES
      TYPE: ascii file
      DESCRIPTION: contains name of languages installed
      CONSTRAINTS: must be readable
                   language names are separated by a NEW LINE
  OUTPUTS:
    ERROR MESSAGES: "A language or 'q' must be entered"
    STATUS MESSAGES: "There are no languages installed"
    HELP MESSAGES: A list of installed languages
    OBJECT: LANGUAGES
      TYPE: ascii file
      CONSTRAINTS: must be writeable
  EFFECT:
    If language = q, return.
    If language = ?, list languages
      in the LANGUAGES file, prompt for language.
      If LANGUAGES file does not exist, print status message("There are
        no languages installed"), prompt for language.
    If language = ' ', print error message("A language or 'q' must
      be entered"), prompt for language.
    If language is not in LANGUAGES file, append language to
      file.
    If ARTS/language directory does not exist, make directory.

SUBACTION: prompt user for compiler.
  INPUTS:
    OBJECT: compiler
      TYPE: character string (keyboard)
      CONSTRAINTS: <=78 ascii characters
                   terminated by RETURN
    OBJECT: COMPILERS
      TYPE: ascii file
      DESCRIPTION: contains name of compilers installed
      CONSTRAINTS: must be readable
                   compiler names are separated by a NEW LINE
    OBJECT: COMPILER_INFO
```

```
        TYPE: ascii file
        DESCRIPTION: contains name of compilers installed, path to
                     compiler and any suffix required
        CONSTRAINTS: must be readable
                     each record is in the form:
                         "name: path:suffix"
                     records are separated by a NEW LINE
    OUTPUTS:
      ERROR MESSAGES: "A compiler or 'q' must be entered"
      STATUS MESSAGES: "There are no compilers installed"
      HELP MESSAGES: A list of compilers installed.
      OBJECT: COMPILERS
        TYPE: ascii file
        CONSTRAINTS: must be writeable
    EFFECT:
      If compiler = q, return.
      If compiler = ?, list compilers in the COMPILERS file,
        prompt for compiler.
      If compiler = ' ', print error message("A compiler or 'q' must
        be entered"), prompt for compiler.
        If COMPILERS file does not exist, print status message("There are
          no compilers installed"), prompt for compiler.
      If compiler is not in COMPILERS file, append compiler
        file.
      If ARTS/language/compiler directory does not exist, make
        directory.
      Check COMPILER_INFO for compiler.

  SUBACTION: prompt user for compiler path
    Note: this action only performed if compiler is not in the
        COMPILER_INFO file.
    INPUTS:
      OBJECT: compiler_path
        TYPE: character string (keyboard)
        CONSTRAINTS: <=78 ascii characters
                     terminated by RETURN
                     file must be readable
    OUTPUTS:
      ERROR MESSAGES: "A path or 'q' must be entered"
                      "Cannot access" compiler_path
      STATUS MESSAGES: "Enter the full UNIX path to the compiler"
      HELP MESSAGES: none
    EFFECT:
      If compiler = q, return.
      If compiler = ?, print status message("Enter the full UNIX path to
        the compiler"), prompt for compiler_path.
    If compiler = ' ', print error message("A path or 'q' must be entered"),
       prompt for compiler_path.
    If compiler_path is not readable, print error message("Cannot access
       compiler_path"), prompt for compiler_path.

SUBACTION: prompt user for suffix required by compiler
  Note: this action only performed if compiler is not in the
       COMPILER_INFO file.
  INPUTS: suffix
    TYPE: character string (keyboard)
    CONSTRAINTS: <= 78 ascii characters
              terminated by RETURN
              suffix required by specified compiler
  OUTPUTS:
    ERROR MESSAGES: none
    STATUS MESSAGES: "Enter any suffix which may be needed"
    HELP MESSAGES: none
    OBJECT: COMPILER_INFO
      TYPE: ascii file
      CONSTRAINTS: must be writeable
  EFFECT:
    If suffix = q, return.
    If suffix = ?, print status message("Enter any suffix which may be
      needed"), prompt for suffix.
    If suffix = ' ', no suffix will be required.
    Append compiler: compiler_path:suffix to COMPILER_INFO.

SUBACTION: prompt user for class.
  INPUTS:
    OBJECT: class
      TYPE: character string (keyboard)
      CONSTRAINTS: <= 78 ascii characters
                terminated by RETURN
    OBJECT: CLASSES
      TYPE: ascii file
      DESCRIPTION: contains name of classes installed
      CONSTRAINTS: must be readable
                class names are separated by a NEW LINE
  OUTPUTS:
    ERROR MESSAGES: "A class or 'q' must be entered"
    STATUS MESSAGES: "There are no classes for this language installed"
    HELP MESSAGES: A list of classes installed.
    OBJECT: CLASSES
      TYPE: ascii file
      CONSTRAINTS: must be writeable
  EFFECT:
    If class = q, return.
    If class = ?, list classes in the CLASSES file,
      prompt for class.
      If CLASSES file does not exist, print status message("There are
        no classes for this language installed"), prompt for class.
    If class = ' ', print error message("A class or 'q' must be entered"),
      prompt for class.
    If class is not in CLASSES file, append to file.
    If ARTS/language/class directory does not exist,
      make directory.

SUBACTION: prompt user for test case file.
  INPUTS:
    OBJECT: tc_path
      TYPE: character string (keyboard)
      CONSTRAINTS: <= 78 ascii characters
                terminated by RETURN
    OBJECT: tc_file
      Note: tc_file = tc_path
      TYPE: ascii file
      CONSTRAINTS: file must be readable
  OUTPUTS:
    ERROR MESSAGES: "A test case file or 'q' must be entered"
                  "Cannot read" test_cases
    STATUS MESSAGES: "Enter name of file which contains your test cases"
    HELP MESSAGES: none
    OBJECT: test_cases.suffix
      TYPE: ascii file
      DESCRIPTION: contains test cases
      CONSTRAINTS: must be writeable
  EFFECT:
    If tc_path = q, return.
    If tc_path = ?, print status message("Enter name of file which
      contains your test cases"), prompt for tc_path.
    If tc_path = ' ', print error message("A test case file or 'q'
      must be entered"), prompt for tc_path.
    If tc_file is not readable, print error message("Cannot read
      tc_path), prompt for tc_path.
    Copy tc_file into ARTS/language/class/test_cases.
    If a suffix was specified move the test_cases file to
      ARTS/language/class/test_cases.suffix.

SUBACTION: prompt user for an input data file.
  INPUTS:
    OBJECT: df_path
      TYPE: character string (keyboard)
```

```
        CONSTRAINTS: <= 78 ascii characters
                terminated by RETURN
      OBJECT: dfile
        Note: dfile = df_path
        TYPE: ascii file
        DESCRIPTION: contains any test data which may be need by
                  the test_cases file.
        CONSTRAINTS: file must be readable
    OUTPUTS:
      ERROR MESSAGES: "Cannot read" df_path
      STATUS MESSAGES: "Enter name of file which contains your input data"
      HELP MESSAGES: none
      OBJECT: input_data
        TYPE: ascii file
        CONSTRAINTS: file must be writeable
    EFFECT:
      If df_path = q, return.
      If df_path = ?, print status message("Enter name of file which
        contains your input data"), prompt for df_path.
      If df_path = ' ', prompt for exec_spec, otherwise,
        If dfile is not readable, print error message("Cannot read
          dfile"), prompt for df_path.
      Copy dfile into ARTS/language/class/input_data.

  SUBACTION: prompt user for execution specification
    Note: this is a line editor which will prompt for
      each line of text.
    INPUTS:
      OBJECT: exec_spec_line
        TYPE: character string (keyboard)
        CONSTRAINTS: max of 10 lines
                each line <= 256 ascii characters
                each line terminated by a RETURN.
        STRUCTURE: <exec_spec> ::=  <c_or_i> <body>
               <c_or_i>   ::=  "$COMPILER" | "$INTERPRETER"
               <body>     ::=  <text> "test_cases"<suffix> <text>
               <suffix>   ::=  ".pl" | ".c" | nil | etc.
               <text>     ::=  any text string excluding "$COMPILER",
                               "$INTERPRETER", "test_cases"<suffix>
                  comment: suffix is determined by compiler or
                        interpreter used.
    OUTPUTS:
      ERROR MESSAGES: "Too many lines"
                "$COMPILER or $INTERPRETER not entered"
                "test_cases.suffix has not been entered"
                "Only one $COMPILER or $INTERPRETER may be entered"
  STATUS MESSAGES: "Enter a line which is needed by the execution
              specification"
  HELP MESSAGES: none
  OBJECT: tmp_file
    TYPE: ascii file
    DESCRIPTION: Editor buffer area
            Exec_spec_line is appended to this file
    CONSTRAINTS: File must be writeable
  OBJECT: Exec_spec
    TYPE: ascii file
    DESCRIPTION: Command script file which is built from tmp_file
    CONSTRAINTS: File must be writeable
EFFECT:
  Open tmp_file for storage of each line of the exec_spec.
  If text = 'q', return.
  If text = '?', print status message("Enter a line which is needed by
      the execution specification").
  If lines of text > 10, print error message("Too many lines"), remove
    the tmp file, prompt for exec_spec.
  If text = '.', check for "$COMPILER" or "$INTERPRETER" and
    "test_cases."suffix.
    If "$COMPILER" or "$INTERPRETER" was not entered, print error
      message("$COMPILER or $INTERPRETER not entered"),
      prompt for a new Exec_spec.
    If "test_cases."suffix was not entered, print error
      message("test_cases.suffix has not been entered"),
      prompt for a new Exec_spec.
    If more than one "$COMPILER" or "$INTERPRETER" was entered, print
      error message("Only one $COMPILER or $INTERPRETER may be entered"),
      prompt for a new Exec_spec.
    Move exec_spec from tmp_file to ARTS/language/class/Exec_spec.
  Append line of text to tmp file.
```

Function INSTALL(exec_spec)

```
SUBACTION: prompt user for language.
  INPUTS:
    OBJECT: language
      TYPE: character string (keyboard)
      CONSTRAINTS: <=78 ascii characters
                terminated by RETURN
    OBJECT: LANGUAGES
      TYPE: ascii file
      DESCRIPTION: contains name of languages installed
      CONSTRAINTS: must be readable
                language names are separated by a NEW LINE
  OUTPUTS:
    ERROR MESSAGES: "A language or 'q' must be entered"
                "The language is not installed"
    STATUS MESSAGES: "There are no languages installed"
    HELP MESSAGES: A list of installed languages
  EFFECT:
    If language = q, return.
    If language = ?, list languages
      in the LANGUAGES file, prompt for language.
      If LANGUAGES file does not exist, print status message("There are
        no languages installed"), prompt for language.
    If language = ' ', print error message("A language or 'q' must
       be entered"), prompt for language.
    If language is not in LANGUAGES file, print error message("The language
      is not installed"), prompt for language.

SUBACTION: prompt user for compiler.
  INPUTS:
    OBJECT: compiler
      TYPE: character string (keyboard)
      CONSTRAINTS: <=78 ascii characters
                 terminated by RETURN
    OBJECT: COMPILERS
      TYPE: ascii file
      DESCRIPTION: contains name of compilers installed
      CONSTRAINTS: must be readable
                compiler names are separated by a NEW LINE
  OUTPUTS:
    ERROR MESSAGES: "A compiler or 'q' must be entered"
                "The compiler is not installed"
    STATUS MESSAGES: "There are no compilers installed"
    HELP MESSAGES: A list of compilers installed.
  EFFECT:
    If compiler = q, return.
    If compiler = ?, list compilers in the COMPILERS file,
      prompt for compiler.
    If compiler = ' ', print error message("A compiler or 'q' must
      be entered"), prompt for compiler.
      If COMPILERS file does not exist, print status message("There are
        no compilers installed"), prompt for compiler.
    If compiler is not in COMPILERS file, print error message("The compiler
      is not installed"), prompt for compiler.

SUBACTION: prompt user for class.
  INPUTS:
    OBJECT: class
      TYPE: character string (keyboard)
      CONSTRAINTS: <= 78 ascii characters
                   terminated by RETURN
    OBJECT: CLASSES
      TYPE: ascii file
      DESCRIPTION: contains name of classes installed
      CONSTRAINTS: must be readable
                 class names are separated by a NEW LINE
  OUTPUTS:
    ERROR MESSAGES: "A class or 'q' must be entered"
                 "The class is not installed"
    STATUS MESSAGES: "There are no classes for this language installed"
    HELP MESSAGES: A list of classes installed.
  EFFECT:
    If class = q, return.
    If class = ?, list classes in the CLASSES file,
      prompt for class.
      If CLASSES file does not exist, print status message("There are
        no classes for this language installed"), prompt for class.
    If class = ' ', print error message("A class or 'q' must be entered"),
      prompt for class.
    If class is not in CLASSES file, print error message("The class
      is not installed"), prompt for class.

SUBACTION: prompt user for execution specification
  Note: this is a line editor which will prompt for
    each line of text.
  INPUTS:
    OBJECT: exec_spec_line
      TYPE: character string (keyboard)
      CONSTRAINTS: max of 10 lines
                 each line <= 256 ascii characters
                 each line terminated by a RETURN.
    STRUCTURE: <exec_spec> ::=  <c_or_i> <body>
               <c_or_i>   ::=  "$COMPILER" | "$INTERPRETER"
               <body>     ::=  <text> "test_cases"<suffix> <text>
               <suffix>   ::=  ".pl" | ".c" | nil | etc.
               <text>     ::=  any text string excluding "$COMPILER",
                               "$INTERPRETER", "test_cases"<suffix>
                  comment: suffix is determined by compiler or
                           interpreter used.
OUTPUTS:
  ERROR MESSAGES: "Too many lines"
                  "$COMPILER or $INTERPRETER not entered"
                  "test_cases.suffix has not been entered"
                  "Only one $COMPILER or $INTERPRETER may be entered"
  STATUS MESSAGES: "Enter a line which is needed by the execution
                   specification"
  HELP MESSAGES: none
  OBJECT: tmp_file
    TYPE: ascii file
    DESCRIPTION: Editor buffer area
                 Exec_spec_line is appended to this file
    CONSTRAINTS: File must be writeable
  OBJECT: Exec_spec
    TYPE: ascii file
    DESCRIPTION: Command script file which is built from tmp_file
    CONSTRAINTS: File must be writeable
EFFECT:
  Open tmp_file for storage of each line of the exec_spec.
  If text = q, return.
  If text = ?, print status message("Enter a line which is needed by
      the execution specification").
  If lines of text > 10, print error message("Too many lines"), remove
    the tmp file, prompt for exec_spec.
  If text = '.', check for "$COMPILER" or "$INTERPRETER" and
    "test_cases."suffix.
    If "$COMPILER" or "$INTERPRETER" was not entered, print error
      message("$COMPILER or $INTERPRETER not entered"),
      prompt for a new Exec_spec.
    If "test_cases."suffix was not entered, print error
      message("test_cases.suffix has not been entered"),
      prompt for a new Exec_spec.
    If more than one "$COMPILER" or "$INTERPRETER" was entered, print
      error message("Only one $COMPILER or $INTERPRETER may be entered"),
      prompt for a new Exec_spec.
    Move exec_spec from tmp_file to ARTS/language/class/Exec_spec.
  Append line of text to tmp file.
```

Function SELECT(test_cases)

```
SUBACTION: prompt user for language.
  INPUTS:
    OBJECT: language
      TYPE: character string (keyboard)
      CONSTRAINTS: <=78 ascii characters
                terminated by RETURN
    OBJECT: LANGUAGES
      TYPE: ascii file
      DESCRIPTION: contains name of languages installed
      CONSTRAINTS: must be readable
               language names are separated by a NEW LINE
  OUTPUTS:
    ERROR MESSAGES: "A language or 'q' must be entered"
               "The language is not installed"
    STATUS MESSAGES: "There are no languages installed"
    HELP MESSAGES: A list of installed languages
  EFFECT:
    If language = q, return.
    If language = ?, list languages
      in the LANGUAGES file, prompt for language.
      If LANGUAGES file does not exist, print status message("There are
        no languages installed"), prompt for language.
    If language = ' ', print error message("A language or 'q' must
      be entered"), prompt for language.
    If language is not in LANGUAGES file, print error message("The language
      is not installed"), prompt for language.

SUBACTION: prompt user for compiler under test.
  INPUTS:
    OBJECT: compiler
      TYPE: character string (keyboard)
      CONSTRAINTS: <=78 ascii characters
               terminated by RETURN
    OBJECT: COMPILERS
      TYPE: ascii file
      DESCRIPTION: contains name of compilers installed
      CONSTRAINTS: must be readable
               compiler names are separated by a NEW LINE
    OBJECT: COMPILER_INFO
      TYPE: ascii file
      DESCRIPTION: contains name of compilers installed, path to
                 compiler and any suffix required
      CONSTRAINTS: must be readable
               each record is in the form:
                    "name: path:suffix"
                records are separated by a NEW LINE
  OUTPUTS:
    ERROR MESSAGES: "A compiler or 'q' must be entered"
                "The compiler is not installed"
    STATUS MESSAGES: "There are no compilers installed"
    HELP MESSAGES: A list of compilers installed.
  EFFECT:
    If compiler = q, return.
    If compiler = ?, list compilers in the COMPILERS file,
      prompt for compiler.
    If compiler = ' ', print error message("A compiler or 'q' must
      be entered"), prompt for compiler.
      If COMPILERS file does not exist, print status message("There are
        no compilers installed"), prompt for compiler.
    If compiler is not in COMPILERS file, print error message("The compiler
      is not installed"), prompt for compiler.

SUBACTION: prompt user for compiler standard.
  INPUTS:
    OBJECT: compiler
      TYPE: character string (keyboard)
      CONSTRAINTS: <=78 ascii characters
                  terminated by RETURN
    OBJECT: COMPILERS
      TYPE: ascii file
      DESCRIPTION: contains name of compilers installed
      CONSTRAINTS: must be readable
                compiler names are separated by a NEW LINE
    OBJECT: COMPILER_INFO
      TYPE: ascii file
      DESCRIPTION: contains name of compilers installed, path to
                    compiler and any suffix required
      CONSTRAINTS: must be readable
                each record is in the form:
                    "name: path:suffix"
                records are separated by a NEW LINE
  OUTPUTS:
    ERROR MESSAGES: "A compiler or 'q' must be entered"
                "The compiler is not installed"
    STATUS MESSAGES: "There are no compilers installed"
    HELP MESSAGES: A list of compilers installed.
  EFFECT:
    If compiler = q, return.
    If compiler = ?, list compilers in the COMPILERS file,
      prompt for compiler.
    If compiler = ' ', print error message("A compiler or 'q' must
      be entered"), prompt for compiler.
      If COMPILERS file does not exist, print status message("There are
        no compilers installed"), prompt for compiler.
    If compiler is not in COMPILERS file, print error message("The compiler
      is not installed"), prompt for compiler.

SUBACTION: prompt user for classes.
  INPUTS:
    OBJECT: class
      TYPE: character string (keyboard)
      CONSTRAINTS: <= 78 ascii characters
                all classes for language selected by "all"
                each class terminated by RETURN
                set of classes terminated by "."
    OBJECT: CLASSES
      TYPE: ascii file
      DESCRIPTION: contains name of classes installed
      CONSTRAINTS: must be readable
                class names are separated by a NEW LINE
  OUTPUTS:
    ERROR MESSAGES: "A class, 'q', or '.' must be entered"
                "The class is not installed"
    STATUS MESSAGES: "There are no classes for this language installed"
    HELP MESSAGES: A list of classes installed.
    OBJECT: exec_paths
      TYPE: ascii file
      DESCRIPTION: Contains the class, the path to the class
                    directory (class_pth), the path to the compiler
                    under test (pth1) and the path to the compiler
                    standard (pth2).
                This is a temporary file which will be removed
                    after execution.
      CONSTRAINTS: must be writeable
                each class to be tested is a record
                each record is in the form:
                    "class: class_pth:pth1:pth2"
                records are separated by a NEW LINE
    OBJECT: exec_spec1
      TYPE: ascii file
      DESCRIPTION: Contains the contents of the Exec_spec for the
                    associated class. $COMPILER or $INTERPRETER
                    is replaced by the path to the compiler under
                    test.
                  File is built in the class directory.
                  This is a temporary file which will be removed
```

```
                    after execution.
        CONSTRAINTS: must be writeable
    OBJECT: exec_spec2
        TYPE: ascii file
        DESCRIPTION: Contains the contents of the Exec_spec for the
                    associated class. $COMPILER or $INTERPRETER
                    is replaced by the path to the compiler standard.
                File is built in the class directory.
                This is a temporary file which will be removed
                    after execution.
        CONSTRAINTS: must be writeable
EFFECT:
    If class = q, return.
    If class = "all", build exec_paths for all installed classes and
                build exec_spec1 and exec_spec2 files for each class.
    If class = ?, list classes in the CLASSES file,
        prompt for class.
        If CLASSES file does not exist, print status message("There are
            no classes for this language installed"), prompt for class.
    If class = ' ', print error message("A class or 'q' must be entered"),
        prompt for class.
    If class is not in CLASSES file, print error message("The class
        is not installed"), prompt for class.
    If class is a valid class, append record to exec_paths and
                build exec_spec1 and exec_spec2 files for the class.
    If class = '.', return to main menu.
```

Function BEGIN_TEST_RUN(exec_paths)

```
SUBACTION: execute test scripts
  INPUTS:
    OBJECT: exec_paths
      TYPE: ascii file
      DESCRIPTION: contains the class, the path to the class
                   directory (class_pth), the path to the compiler
                   under test (pth1) and the path to the compiler
                   standard (pth2).
                this is a temporary file which will be removed
                   after execution.
      CONSTRAINTS: must be readable.
                each class to be tested is a record.
                each record is in the form:
                   "class: class_pth:pth1:pth2".
                records are separated by a NEW LINE.
    OBJECT: exec_spec1
      TYPE: ascii file
      DESCRIPTION: Contains the contents of the Exec_spec for the
                   associated class. $COMPILER or $INTERPRETER
                   has been replaced by the path to the compiler
                   under test.
                 File located is in the class directory.
                 This is a temporary file which will be removed
                   after execution.
      CONSTRAINTS: must be readable
    OBJECT: exec_spec2
      TYPE: ascii file
      DESCRIPTION: Contains the contents of the Exec_spec for the
                   associated class. $COMPILER or $INTERPRETER
                   has been replaced by the path to the compiler
                   standard.
                File located in the class directory.
                This is a temporary file which will be removed
                   after execution.
      CONSTRAINTS: must be readable
  OUTPUTS:
    ERROR MESSAGES: "Cannot read exec_paths"
                "Improper exec_paths format"
                "Cannot change directory to directory path"
                "Cannot access exec_spec1"
                "Cannot access exec_spec2"
    STATUS MESSAGES: "Running tests for class: class"
    OBJECT: output_data
      TYPE: ascii file
      DESCRIPTION: Contains the results of executing the the exec_spec
                   files.
      CONSTRAINTS: Must be writeable.
               File located in the class/compiler directory.
               There will be two files per class, one for the
                   compiler under test, one for the compiler standard.
  EFFECT:
    If exec_paths cannot be read, then print error message("Cannot read
      exec_paths"), return to main menu.
    LOOP:
      Get a record from exec_paths until no records remain.
        If there is no ": " in the record, then print error message("Improper
          exec_paths format"), return to main menu.
        Get the class name.
        Get the path to the class and change to that directory.
          If cannot change directory, print error message("Cannot change
            directory to directory path"), return to
            main menu.
        If exec_spec1 is not in the directory, print error message("Cannot access
            exec_spec1"), continue.
        If exec_spec2 is not in the directory, print error message("Cannot access
            exec_spec2"), continue.
        Print_msg("Running tests for class: class").
        Build command line to execute exec_spec1 and trap output in
          output_data file (located in compiler under test directory).
        Execute command line.
        Build command line to execute exec_spec2 and trap output in
          output_data file (located in compiler standard directory).
        Execute command line.
        Change directory to home directory.

SUBACTION: analyze results
  INPUTS:
    OBJECT: exec_paths
      TYPE: ascii file
      DESCRIPTION: Contains the class, the path to the class
                   directory (class_pth), the path to the compiler
                   under test (pth1) and the path to the compiler
                   standard (pth2).
               This is a temporary file which will be removed
                   after execution.
      CONSTRAINTS: must be readable.
               Each class to be tested is a record.
               Each record is in the form:
                   "class: class_pth:pth1:pth2".
               Records are separated by a NEW LINE.
```

```
 OBJECT: output_data
   TYPE: ascii file
   DESCRIPTION: Contains the results of executing the the exec_spec
            files.
   CONSTRAINTS: Must be writeable.
          File located in the class/compiler directory.
          There will be two files per class, one for the
            compiler under test, one for the compiler standard
OUTPUTS:
 ERROR MESSAGES: "Cannot read exec_paths"
          "Improper exec_paths format"
 STATUS MESSAGES: "Completed analysis of data"
 OBJECT: test_analysis
   TYPE: ascii file
   DESCRIPTION: Contains the testing analysis.
          First line is the user who performed the tests.
          Second line is the date.
          Third line is the language.
          Successive lines are the class being tested, the
            compilers used, and a "diff" of the outputs.
   CONSTRAINTS: Must be writeable.
          File created in the user's rje directory.
 OBJECT: analysis
   TYPE: ascii file
   DESCRIPTION: Contains the results of performing a "diff" on
            two output_data files.
   CONSTRAINTS: Must be writeable.
          File created in the compiler directory of the
            compiler under test.
 OBJECT: header
   TYPE: ascii file
   DESCRIPTION: Maintains a record of the following information:
            - Person doing the testing
            - date of testing
            - compiler under test
            - compiler standard
            - pass/fail status of test run
   CONSTRAINTS: Must be writeable
          File located in the class directory
EFFECT:
 If exec_paths cannot be read, then print error message("Cannot read
   exec_paths"), return to main menu.
 Write person performing tests and date into test_analysis.
 LOOP:
   Get a record from exec_paths until no records remain.
     If there is no ":" in the record, then print error message("Improper
        exec_paths format"), return to main menu.
    Get the class name.
    Get the path to the class.
    Get the path to both compilers.
    Build command line to do a "diff" of the two output data
        files for the associated class and direct output to analysis
        file.
    Execute command line.
    If analysis file was created, then a difference was detected. Write
        "FAILED" into test_analysis.
    If analysis file was not created, then no differences were
        detected. Write "PASSED" into test_analysis.
    Update the header file with applicable data.
    Write the language being used,.the compilers being tested, the
        class, the status, and any differences in output into
        test_analysis file.
Print_sts("Completed analysis of data").
Return to main menu.
```

Function REPORT(test_analysis)

```
ACTION: print analysis of testing results
INPUTS:
   OBJECT: test_analysis
      TYPE: ascii file
      DESCRIPTION: Contains the testing analysis.
                  First line is the user who performed the tests.
                  Second line is the date.
                  Third line is the language.
                  Successive lines are the class being tested, the
                      compilers used, and a "diff" of the outputs.
      CONSTRAINTS: Must be readable.
                  File located in the user's rje directory.
OUTPUTS:
   STATUS MESSAGE: "The report has been sent to the line printer"
EFFECT: Create a command line to send the test_analysis file to
           the printer.
        Execute the command line.
        Print_sts("The report has been sent to the line printer").
        Return to main menu.
```

Function RETRIEVE(class)

```
SUBACTION: prompt user for language.
  INPUTS:
    OBJECT: language
      TYPE: character string (keyboard)
      CONSTRAINTS: <=78 ascii characters
               terminated by RETURN
    OBJECT: LANGUAGES
      TYPE: ascii file
      DESCRIPTION: contains name of languages installed
      CONSTRAINTS: must be readable
               language names are separated by a NEW LINE
  OUTPUTS:
    ERROR MESSAGES: "A language or 'q' must be entered"
               "The language is not installed"
    STATUS MESSAGES: "There are no languages installed"
    HELP MESSAGES: A list of installed languages
  EFFECT:
    If language = q, return.
    If language = ?, list languages
      in the LANGUAGES file, prompt for language.
      If LANGUAGES file does not exist, print status message("There are
        no languages installed"), prompt for language.
    If language = ' ', print error message("A language or 'q' must
       be entered"), prompt for language.
    If language is not in LANGUAGES file, print error message("The language
      is not installed"), prompt for language.

SUBACTION: prompt user for compiler
  INPUTS:
    OBJECT: compiler
      TYPE: character string (keyboard)
      CONSTRAINTS: <=78 ascii characters
               terminated by RETURN
    OBJECT: COMPILERS
      TYPE: ascii file
      DESCRIPTION: contains name of compilers installed
      CONSTRAINTS: must be readable
               compiler names are separated by a NEW LINE
  OUTPUTS:
    ERROR MESSAGES: "A compiler or 'q' must be entered"
               "The compiler is not installed"
    STATUS MESSAGES: "There are no compilers installed"
    HELP MESSAGES: A list of compilers installed.
  EFFECT:
    If compiler = q, return.
    If compiler = ?, list compilers in the COMPILERS file,
      prompt for compiler.
    If compiler = ' ', print error message("A compiler or 'q' must
      be entered"), prompt for compiler.
      If COMPILERS file does not exist, print status message("There are
        no compilers installed"), prompt for compiler.
    If compiler is not in COMPILERS file, print error message("The compiler
      is not installed"), prompt for compiler.

SUBACTION: prompt user for class.
  INPUTS:
    OBJECT: class
      TYPE: character string (keyboard)
      CONSTRAINTS: <= 78 ascii characters
                Terminated by a RETURN.
    OBJECT: CLASSES
      TYPE: ascii file
      DESCRIPTION: contains name of classes installed
      CONSTRAINTS: must be readable
                class names are separated by a NEW LINE
    OBJECTS: All files in the language/class and compiler/class
            directories for the specified language, compiler and class
      TYPE: ascii files
      DESCRIPTION: contains data associated with the class
      CONSTRAINTS: must be readable
  OUTPUTS:
    ERROR MESSAGES: "A class, or 'q' must be entered"
                "The class is not installed"
    STATUS MESSAGES: "There are no classes for this language installed"
                "All data files have been copied to your rje
                  directory"
    HELP MESSAGES: A list of classes installed.
    OBJECTS: All files in the language/class and compiler/class
            directories for the specified language, compiler and class.
      TYPE: ascii files
      DESCRIPTION: contains data associated with the class
      CONSTRAINTS: writeable into the user's rje directory.
  EFFECT:
    If class = q, return to main menu.
    If class = ?, list classes in the CLASSES file,
      prompt for class.
      If CLASSES file does not exist, print status message("There are
        no classes for this language installed"), prompt for class.
    If class = ' ', print error message("A class or 'q' must be entered"),
      prompt for class.
```

```
If class is not in CLASSES file, print error message("The class
  is not installed"), prompt for class.
Build command lines to copy all data files for the specified class
  into user's rje directory.
Execute command line.
Print_sts("All data files have been copied to your rje directory")
Return to main menu.
```

Function DELETE(class)

```
SUBACTION: prompt user for language.
  INPUTS:
    OBJECT: language
      TYPE: character string (keyboard)
      CONSTRAINTS: <=78 ascii characters
                terminated by RETURN
    OBJECT: LANGUAGES
      TYPE: ascii file
      DESCRIPTION: contains name of languages installed
      CONSTRAINTS: must be readable
                language names are separated by a NEW LINE
  OUTPUTS:
    ERROR MESSAGES: "A language or 'q' must be entered"
                "The language is not installed"
    STATUS MESSAGES: "There are no languages installed"
    HELP MESSAGES: A list of installed languages
  EFFECT:
    If language = q, return.
    If language = ?, list languages
      in the LANGUAGES file, prompt for language.
      If LANGUAGES file does not exist, print status message("There are
        no languages installed"), prompt for language.
    If language = ' ', print error message("A language or 'q' must
       be entered"), prompt for language.
    If language is not in LANGUAGES file, print error message("The language
      is not installed"), prompt for language.

SUBACTION: prompt user for compiler
  INPUTS:
    OBJECT: compiler
      TYPE: character string (keyboard)
      CONSTRAINTS: <=78 ascii characters
                terminated by RETURN
    OBJECT: COMPILERS
      TYPE: ascii file
      DESCRIPTION: contains name of compilers installed
      CONSTRAINTS: must be readable
                compiler names are separated by a NEW LINE
  OUTPUTS:
    ERROR MESSAGES: "A compiler or 'q' must be entered"
                "The compiler is not installed"
    STATUS MESSAGES: "There are no compilers installed"
    HELP MESSAGES: A list of compilers installed.
  EFFECT:
    If compiler = q, return.
    If compiler = ?, list compilers in the COMPILERS file,
      prompt for compiler.
    If compiler = ' ', print error message("A compiler or 'q' must
      be entered"), prompt for compiler.
      If COMPILERS file does not exist, print status message("There are
        no compilers installed"), prompt for compiler.
    If compiler is not in COMPILERS file, print error message("The compiler
      is not installed"), prompt for compiler.

SUBACTION: prompt user for class.
  INPUTS:
    OBJECT: class
      TYPE: character string (keyboard)
      CONSTRAINTS: <= 78 ascii characters
                Terminated by a RETURN.
    OBJECT: CLASSES
      TYPE: ascii file
      DESCRIPTION: contains name of classes installed
      CONSTRAINTS: must be readable
                 class names are separated by a NEW LINE
  OUTPUTS:
    ERROR MESSAGES: "A class, or 'q' must be entered"
                "The class is not installed"
    STATUS MESSAGES: "There are no classes for this language installed"
                 "All data files have been copied to your rje
                   directory"
    HELP MESSAGES: A list of classes installed.
    OBJECT: CLASSES
      TYPE: ascii file
      DESCRIPTION: contains name of classes installed
      CONSTRAINTS: must be writeable
                class names are separated by a NEW LINE
    OBJECTS: All files in the language/class and compiler/class
             directories for the specified language, compiler
             and class.
      TYPE: ascii files
      DESCRIPTION: contains data associated with the class
      CONSTRAINTS: Must be writeable.
  EFFECT:
    If class = q, return to main menu.
    If class = ?, list classes in the CLASSES file,
      prompt for class.
      If CLASSES file does not exist, print status message("There are
        no classes for this language installed"), prompt for class.
    If class = ' ', print error message("A class or 'q' must be entered"),
   prompt for class.
If class is not in CLASSES file, print error message("The class
   is not installed"), prompt for class.
Build command lines to remove all data files for the specified class
Execute command line.
Return to main menu.
```

Function HELP(helpfile)

```
ACTION: display contents of helpfile
INPUTS:
  OBJECT: helpfile
    TYPE: ascii file
    DESCRIPTION: Contains any helpful information for
                 the general operation of the system.
    CONSTRAINTS: Must be readable.
            File located in the root directory of ARTS
EFFECT:
  LOOP:
    Paginate the display by reading 20 lines of the
      helpfile file into the display buffer.
    A space reads in another 20 line buffer.
    A RETURN reads in a single line.
    A 'q' returns to the main menu.
```

Function DISPLY(test_analysis)

```
ACTION: display analysis of testing results
INPUTS:
  OBJECT: test_analysis
    TYPE: ascii file
    DESCRIPTION: Contains the testing analysis.
               First line is the user who performed the tests.
               Second line is the date.
               Third line is the language.
               Successive lines are the class being tested, the
                 compilers used, and a "diff" of the outputs.
    CONSTRAINTS: Must be readable.
               File located in the user's rje directory.
EFFECT:
  LOOP:
    Paginate the testing results by reading 20 lines of the
      test_analysis file into the display buffer.
    A space reads in another 20 line buffer.
    A RETURN reads in a single line.
    A 'q' returns to the main menu.
```

APPENDIX B

ARTS Source Code

```
#########################################################
#
# ARTS - Automated Regression Test System
#
# Execute "make" to build ARTS
# Note: LIB must be set to -ltermlib for BSD machines
#
#########################################################

OFILES= main.o buildwin.o mainw.o installw.o deletew.o
    retrievw.o selectw.o execute.o more.o function.o
    report.o install.o retrieve.o delete.o select.o
    post_exec.o

CFILES= main.c buildwin.c mainw.c installw.c deletew.c
    retrievw.c selectw.c execute.c more.c function.c
    report.c install.c retrieve.c delete.c select.c
    post_exec.c

HFILES= arts.h header.h

# Set LIB to -ltermlib for BSD machines
LIB=-lcurses

arts:     $(OFILES)
   cc -x -o arts $(OFILES) $(LIB)

main.o:     main.c $(HFILES)
   cc -s -c -w main.c

mainw.o:   mainw.c $(HFILES)
   cc -s -c -w mainw.c

buildwin.o:   buildwin.c $(HFILES)
   cc -s -c -w buildwin.c

installw.o:   installw.c $(HFILES)
   cc -s -c -w installw.c

retrievw.o:   retrievw.c $(HFILES)
   cc -s -c -w retrievw.c

selectw.o:   selectw.c $(HFILES)
   cc -s -c -w selectw.c
```

```
deletew.o:   deletew.c $(HFILES)
   cc -s -c -w deletew.c

more.o:      more.c $(HFILES)
   cc -s -c -w more.c

execute.o:   execute.c $(HFILES)
   cc -s -c -w execute.c

function.o:   function.c $(HFILES)
   cc -s -c -w function.c

report.o:   report.c $(HFILES)
   cc -s -c -w report.c

install.o:   install.c $(HFILES)
   cc -s -c -w install.c

delete.o:   delete.c $(HFILES)
   cc -s -c -w delete.c

retrieve.o:   retrieve.c $(HFILES)
   cc -s -c -w retrieve.c

select.o:   select.c $(HFILES)
   cc -s -c -w select.c

post_exec.o:   post_exec.c $(HFILES)
   cc -s -c -w post_exec.c
```

```
/****************************************************
*
* arts.h
*
* This header file is common to all source files
*
* The PRINT_CMD variable may be changed to suit the
* local environment.
*
****************************************************/

#include <stdio.h>
#include <signal.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/dir.h>
#include <sys/stat.h>
#include <curses.h>

#define tmpfile "/tmp/artsXXXXXX"
#define LINESZ 80
#define PRINT_CMD "opr"

int onintrpt();


/*****************************************************
*
* header.h
*
* This header file is common to all source files except
* main.c
*
*****************************************************/

extern WINDOW *mainwin;      /* main menu window */
extern WINDOW *instalwin;  /* installation form window */
extern WINDOW *deletewin;  /* deletion form window */
extern WINDOW *retrievewin;     /* retrieval form window */
extern WINDOW *selectwin;  /* selection form window */
extern WINDOW *contentwin;      /* contents form window */
extern WINDOW *errorwin;   /* error message window */
extern WINDOW *statuswin;  /* status message window */
extern WINDOW *scrolwin;   /* scrolling window */
extern char langpth[];
extern char comp_pth[];
extern char comp_pth2[];
extern char classpth1[];
extern char classpth2[];
extern char exec_paths[];

extern char *root;
extern char *home;
```

```
/****************************************************
*
* main.c - ARTS main module
*
* Sets various path variables and initializes the windows.
*
****************************************************/

#include "arts.h"

/* Funny characters which can be used in a path name */
char *allowc = "!$%^&-_+= :.";

char lang[LINESZ];
char langpth[LINESZ];       /* root/langdir */
char compiler[LINESZ];
char comp_pth[LINESZ];      /* root/langdir/compdir */
char comp_pth2[LINESZ];     /* compiler standard */
char class[LINESZ];
char classpth1[256];   /* root/langdir/clsdir */
char classpth2[256];   /* root/langdir/compdir/clsdir */
char exec_paths[LINESZ];   /* execution paths file */

char *getenv();
char *root;   /* ARTS file system root */
char *home;   /* user's home directory */
char *strchr();

int Row, Col;

WINDOW *mainwin;        /* main menu window */
WINDOW *instalwin;      /* installation form window */
WINDOW *deletewin;      /* deletion form window */
WINDOW *retrievewin;       /* retrieval form window */
WINDOW *selectwin;      /* selection form window */
WINDOW *errorwin;       /* error message window */
WINDOW *statuswin;      /* status message window */
WINDOW *scrolwin;       /* scrolling (overlayed) window */

main()

{
   FILE *fptr;
   char resp;

   if((root = getenv("ARTS")) == NULL)
```

```
    {
        printf("ARTS path is not set\n");
        exit(1);
    }

    if((home = getenv("HOME")) == NULL)
    {
        printf("HOME path is not set\n");
        exit(1);
    }

    sprintf(exec_paths,"%s/exec_paths",root);
    if((access(exec_paths,0)) == 0)
        unlink(exec_paths);

    signal(SIGINT,onintrpt);
    signal(SIGHUP,onintrpt);

    initscr();  /* Standard Curses initialization */
    scrolwin = newwin(22,COLS,0,0);        /* create overlay window */
    errorwin = newwin(1,COLS,LINES - 1,0);  /* create error message window */
    statuswin = newwin(1,COLS,LINES - 2,0);  /* create status message window */
    mainwin = newwin(22,COLS,0,0);          /* creat the main window */
    instalwin = newwin(20,COLS,0,0);        /* create the installation window */
    deletewin = newwin(11,COLS,5,0);        /* create the deletion window */
    retrievewin = newwin(11,COLS,5,0);  /* create the deletion window */
    selectwin = newwin(21,COLS,1,0);        /* create the deletion window */

    mainw();  /* call the main menu */
}
```

```
/***************************************************
*
* buildwin.c - build windows used by ARTS
*
* This module contains the functions bldmainwin,
* bldinstalwin, blddeletewin, bldretrievewin and
* bldselectwin
*
****************************************************/

#include "arts.h"
#include "header.h"

bldmainwin()  /* build the main window */
{
    wmove(mainwin,4,COLS - 50);
    waddstr(mainwin,"ARTS MAIN MENU");
    wmove(mainwin,7,COLS - 75);
    waddstr(mainwin,"TAB the cursor to the desired option");
    wmove(mainwin,8,COLS - 75);
    waddstr(mainwin,"Enter 'x' to select the option");
    wmove(mainwin,10,COLS - 65);
    waddstr(mainwin,"[ ] Install test cases");
    wmove(mainwin,11,COLS - 65);
    waddstr(mainwin,"[ ] Select test cases for execution");
    wmove(mainwin,12,COLS - 65);
    waddstr(mainwin,"[ ] Begin test run");
    wmove(mainwin,13,COLS - 65);
    waddstr(mainwin,"[ ] Install execution specification");
    wmove(mainwin,14,COLS - 65);
    waddstr(mainwin,"[ ] Display testing results");
    wmove(mainwin,15,COLS - 65);
    waddstr(mainwin,"[ ] Generate report of testing results");
    wmove(mainwin,16,COLS - 65);
    waddstr(mainwin,"[ ] Retrieve test case data files");
    wmove(mainwin,17,COLS - 65);
    waddstr(mainwin,"[ ] Delete class of test cases");
    wmove(mainwin,18,COLS - 65);
    waddstr(mainwin,"[ ] Help");
    wmove(mainwin,19,COLS - 65);
    waddstr(mainwin,"[ ] Exit ARTS");
}

bldinstalwin()  /* build the installation form */
{
    box(instalwin,'*','*');
    wmove(instalwin,1,29);
    waddstr(instalwin,"ARTS INSTALLATION FORM");
    wmove(instalwin,4,5);
    waddstr(instalwin,"Enter the data and press RETURN
        ('q' to quit, '?' for help)");
}

blddeletewin()   /* build the deletion form */
{
    box(deletewin,'*','*');
    wmove(deletewin,1,29);
    waddstr(deletewin,"ARTS DELETION FORM");
    wmove(deletewin,4,5);
    waddstr(deletewin,"Enter the data and press RETURN
        ('q' to quit, '?' for help)");
}

bldretrievewin()   /* build the retrieval form */
{
    box(retrievewin,'*','*');
    wmove(retrievewin,1,29);
    waddstr(retrievewin,"ARTS RETRIEVAL FORM");
    wmove(retrievewin,4,5);
    waddstr(retrievewin,"Enter the data and press RETURN
        ('q' to quit, '?' for help)");
}

bldselectwin()   /* build the selection form */
{
    box(selectwin,'*','*');
    wmove(selectwin,1,29);
    waddstr(selectwin,"ARTS SELECTION FORM");
    wmove(selectwin,4,5);
    waddstr(selectwin,"TAB the cursor to the desired field");
    wmove(selectwin,5,5);
    waddstr(selectwin,"Enter the data and press RETURN
        ('q' to quit, '?' for help)");
}
```

```
/*************************************************************
*
*
* delete.c - delete files and directories associated with
*       a given class.
*
* This module contains the function dclas.
* All data files for
* the "class" argument are deleted.
* The class directories are also deleted.
*
* This function expects to have the langpth already set.
*
* If an error is detected, a "1" is
* returned; otherwise a "0" is returned.
*************************************************************/

#include "arts.h"
#include "header.h"

dclas(class)
char *class;
{
   FILE *rptr;
   FILE *wptr;
   char buf[256];
   char dir[LINESZ];
   char rpath[LINESZ];  /* read path */
   char wpath[LINESZ];  /* write path */
   char line[LINESZ];
   int got_class;

   lower(class);  /* convert all classs to lower case */
   strcpy(dir,class);
   convert(dir);  /* convert class name to directory name */

   sprintf(rpath,"%s/CLASSES",langpth);
   if((rptr = fopen(rpath,"r"))==NULL) /* no class file exists */
   {
      errormsg("No class file exists for the language");
      return(1);
   }

   /* classs file exists */
   sprintf(wpath,"%s/newclass",langpth);
   signal(SIGINT,SIG_IGN);
```

```
wptr = fopen(wpath,"w");
got_class = 0;
while((fgets(line,LINESZ,rptr)) != NULL)
{
   line[strlen(line) - 1] = '\0';   /* get rid of NL */
   if(strcmp(line,class) == 0)   /* class is in db */
   {
      sprintf(buf,"rm -fr %s/%s",langpth,dir);
      system(buf);   /* remove the class directory */
      got_class = 1;
   }
   else
      fprintf(wptr,"%s\n",line);
}
fclose(rptr);
fclose(wptr);
if(!got_class)   /* class was not found in class file */
{
   sprintf(buf,"The class %s is not installed for the language",class);
   unlink(wpath);
   signal(SIGINT,onintrpt);
   errormsg(buf);
   return(1);
}

/* create a new class file with the deleted class removed */
sprintf(buf,"mv %s %s",wpath,rpath);
system(buf);

/* We now need to get each compiler path and delete the */
/* directory for the associated class */

sprintf(rpath,"%s/COMPILERS",langpth);
if((rptr = fopen(rpath,"r"))==NULL) /* no compiler file exists */
{
   errormsg("No compilers file exists for the language");
   signal(SIGINT,onintrpt);
   return(1);
}
while((fgets(line,LINESZ,rptr)) != NULL)
{
   line[strlen(line) - 1] = '\0';   /* get rid of NL */
   sprintf(buf,"rm -fr %s/%s/%s",langpth,line,dir);
   system(buf);
}
fclose(rptr);
```

```
    signal(SIGINT,onintrpt);
    return(0);
}
```

```
#include "arts.h"
#include "header.h"

/***********************************************************
*
*
* deletew.c - ARTS deletion window
*
* Get the language, compiler and class to be
* deleted and call dclas to delete all files
* and directories associated with the class
*
************************************************************/

deletew()
{
   FILE *fptr;
   char resp[LINESZ];
   char buf[LINESZ];
   int row;

   blddeletewin();  /* build the deletion window */
   cbreak();
   nl();
   noecho();
   wclear(mainwin);
   wrefresh(mainwin);
   for(;;)  /* get the language */
   {
      wmove(deletewin,7,2);
      waddstr(deletewin,"Enter language: ");
      wclrtoeol(deletewin);
      box(deletewin,'*','*');
      wrefresh(deletewin);
      mywgetstr(deletewin,resp);
      clearerror();
      if(resp[0] == 'q' && resp[1] == '\0')   /* return to main menu */
         return;
      else if(resp[0] == '?' && resp[1] == '\0') /* list languages */
      {
         sprintf(buf,"%s/LANGUAGES",root);
         if((fptr = fopen(buf,"r")) == NULL)
         {
            statusmsg("There are no languages installed");
            continue;
         }
```

```
        overwrite(deletewin,mainwin);
        wclear(deletewin);
        wmove(deletewin,1,5);
        waddstr(deletewin,"Installed languages:");
        row = 3;
        while((fgets(buf,LINESZ,fptr)) != NULL)
        {
            wmove(deletewin,row++,1);
            waddstr(deletewin,buf);
        }
        fclose(fptr);
        row++;
        wmove(deletewin,row++,5);
        waddstr(deletewin,"Press any key to continue");
        wrefresh(deletewin);
        wgetch(deletewin);
        overwrite(mainwin,deletewin);
        wclear(mainwin);
        wrefresh(deletewin);
        continue;
    }
    else if(resp[0] == '\0')   /* RETURN was entered */
    {
        errormsg("A language or 'q' must be entered");
        continue;
    }
    if((glang(resp)) != 0)   /* retrieve the language */
        continue;   /* A failure occurred */
    break;
}

for(;;)   /* get the class */
{
    wmove(deletewin,8,2);
    waddstr(deletewin,"Enter class: ");
    wclrtoeol(deletewin);
    box(deletewin,'*','*');
    wrefresh(deletewin);
    mywgetstr(deletewin,resp);
    clearerror();
    if(resp[0] == 'q' && resp[1] == '\0')   /* return to main menu */
        return;
    else if(resp[0] == '?' && resp[1] == '\0') /* list classes */
    {
        sprintf(buf,"%s/CLASSES",langpth);
        if((fptr = fopen(buf,"r")) == NULL)
```

```
            {
               statusmsg("There are no classes for this language installed");
               continue;
            }
            overwrite(deletewin,mainwin);
            wclear(deletewin);
            wmove(deletewin,1,5);
            waddstr(deletewin,"Installed classes:");
            row = 3;
            while((fgets(buf,LINESZ,fptr)) != NULL)
            {
               wmove(deletewin,row++,1);
               waddstr(deletewin,buf);
            }
            fclose(fptr);
            row++;
            wmove(deletewin,row++,5);
            waddstr(deletewin,"Press any key to continue");
            wrefresh(deletewin);
            wgetch(deletewin);
            overwrite(mainwin,deletewin);
            wclear(mainwin);
            wrefresh(deletewin);
            continue;
         }
         else if(resp[0] == '\0')   /* RETURN was entered */
         {
            errormsg("A class or 'q' must be entered");
            continue;
         }
         if((dclas(resp)) != 0)   /* delete the class */
            continue;   /* A failure occurred */
         break;
      }

      sprintf(buf,"The class %s has been deleted",resp);
      statusmsg(buf);
   }
```

```
/*******************************************************
*
* execute.c - start the execution of a test run
*
* This function requires an exec_paths file to have
* been previously built by the select function.
*
* The .exec_spec1 and .exec_spec2 files are used as
* execution scripts for the shell. The shell executes
* these scripts and redirects all output to the
* output_data file for both compilers. The .exec_spec
* files are removed after execution.
*
* The post_exec routine is called after execution of
* the scripts.
*
* A "1" is returned on a failure from this function;
* otherwise, a "0" is returned.
*
*******************************************************/

#include "arts.h"
#include "header.h"
#include <pwd.h>
execute()
{
    struct stat stbuf;
    FILE *rptr;
    char line[256];
    char cur_dir[LINESZ];
    char class[LINESZ];
    char exec_pth[LINESZ];  /* path to the .exec_spec dir */
    char c_u_tst[LINESZ];   /* path to comp_under_tst dir */
    char c_std[LINESZ];     /* path to comp_std dir */

    /* Get the current directory. Ugh!!!!!!! */
    strcpy(line,"pwd > /tmp/ARTS");
    system(line);
    rptr = fopen("/tmp/ARTS","r");
    fgets(cur_dir,LINESZ,rptr);
    fclose(rptr);
    unlink("/tmp/ARTS");
    if((rptr = fopen(exec_paths,"r")) == NULL)
    {
        sprintf(line,"Cannot read %s",exec_paths);
        errormsg(line);
    return(1);
}
while((fgets(line,256,rptr)) != NULL)
{
    /* line[strlen(line) -1] = '\0'; */
    if(strchr(line,':') == NULL)
    {
        errormsg("Improper exec_paths format");
        fclose(rptr);
        return(1);
    }
    strcpy(class,strtok(line,":"));
    strcpy(exec_pth,strtok(NULL,":"));
    strcpy(c_u_tst,strtok(NULL,":"));
    strcpy(c_std,strtok(NULL,"\n"));
    if(chdir(exec_pth) != 0)
    {
        fclose(rptr);
        sprintf(line,"Cannot chdir to %s",exec_pth);
        errormsg(line);
        return(1);
    }
    if(stat(c_u_tst, &stbuf) == -1)   /* directory does not exist */
    {
        signal(SIGINT,SIG_IGN);
        sprintf(line,"mkdir %s",c_u_tst);
        system(line);
        signal(SIGINT,onintrpt);
    }
    if(stat(c_std, &stbuf) == -1)   /* directory does not exist */
    {
        signal(SIGINT,SIG_IGN);
        sprintf(line,"mkdir %s",c_std);
        system(line);
        signal(SIGINT,onintrpt);
    }
    if((access(".exec_spec1",0)) != 0)
    {
        strcpy(line,"Cannot access .exec_spec1, continuing");
        errormsg(line);
        continue;
    }
    sprintf(line,"Running tests for class: %s...",class);
    statusmsg(line);
    sprintf(line,"sh .exec_spec1 1> %s/output_data 2>&1",c_u_tst);
    system(line);
```

```
            unlink(".exec_spec1");
            if((access(".exec_spec2",0)) != 0)
            {
                strcpy(line,"Cannot access .exec_spec2, continuing");
                errormsg(line);
                continue;
            }
            sprintf(line,"sh .exec_spec2 1> %s/output_data 2>&1",c_std);
            system(line);
            unlink(".exec_spec2");
        }
        fclose(rptr);
        chdir(cur_dir);
        post_exec();    /* Call the post execution routine */
        return(0);
    }
```

```
/****************************************************
*
* function.c - contains various miscellaneous functions
*
****************************************************/

#include "arts.h"
#include "header.h"

/****************************************************
* Get a string from the window. The wgetstr function
* provided with Curses did not do what I wanted it to
* do, so I had to write my own.
****************************************************/

mywgetstr(win,buf)

WINDOW *win;
char *buf;
{
   char *p;
   int c;
   int y, x;
   int old_y, old_x;
   int i = 0;
   int erasec, killc;

   erasec = erasechar();
   killc = killchar();
   getyx(win,old_y,old_x);
   getyx(win,y,x);
   while((c = wgetch(win)) != '\n')
   {
      if(c == erasec)
      {
         if(old_x < x)
         {
            wmove(win,y,--x);
            wclrtoeol(win);
            box(win,'*','*');
            buf--;
            wrefresh(win);
         }
      }
      else if(c == killc)
      {
            while(old_x < x)
            {
                wmove(win,y,--x);
                buf--;
            }
            wclrtoeol(win);
            box(win,'*','*');
            wrefresh(win);
        }
        else
        {
            waddch(win,c);
            x++;
            *buf = c;
            buf++;
            wrefresh(win);
        }
    }
    *buf = '\0';
}

errormsg(str)   /* Display a message in the error message
window */
char *str;

{
    wstandout(errorwin);
    wmove(errorwin,0,0);
    wclrtoeol(errorwin);
    wrefresh(errorwin);
    waddstr(errorwin,str);
    wrefresh(errorwin);
    putchar(7);
}

statusmsg(str)   /* Display a message in the status window */
char *str;

{
    wstandout(statuswin);
    wmove(statuswin,0,0);
    wclrtoeol(statuswin);
    wrefresh(statuswin);
    waddstr(statuswin,str);
    wrefresh(statuswin);
    putchar(7);
```

```
}

clearstat()  /* Clear the status window */
{
   wmove(statuswin,0,0);
   wclrtoeol(statuswin);
   wrefresh(statuswin);
}

clearerror()  /* Clear the error message window */
{
   wmove(errorwin,0,0);
   wclrtoeol(errorwin);
   wrefresh(errorwin);
}

/*
  Directory paths for each language will be built using the language
  as a directory name. No wild-card or white-space characters or allowed.
  These characters will be substituted for a dash.
  The maximum length of a name is limited to 14 characters.
*/
convert(buf)
char *buf;
{
   extern char *allowc;
   char *p1, *p2;
   int i;
   p1 = buf;
   for(i=0;*p1 != '\0' && i != 13;p1++,i++)
   {
      p2 = allowc;
      /* Test for allowable characters */
      if(isalnum(*p1))
         continue;
      while(*p2++ != '\0')
      {
         if(*p2 == *p1)
            break;
      }
      if(*p2 == '\0')  /* the character was not found */
         *p1 = '_';
   }
   *p1 = '\0';
}
```

```
lower(buf)   /* Convert a string to lower case characters */
char *buf;
{
   char *p1;
   char c;
   p1 = buf;
   while(*p1 != '\0')
   {
      if(isupper(*p1))   /* convert only upper case letters */
      {
         c = tolower(*p1);
         *p1 = c;
      }
      p1++;
   }
}

/*****************************************************************
* This functions checks str1 to determine if the pattern str2
* is in it. It returns a 0 if yes, a 1 if no.
*****************************************************************/

match(str1,str2)
char *str1, *str2;

{
   char *p1, *p2, *p3;

   p1 = str1;
   while(*p1)
   {
      if(*p1 == *str2)
      {
         p2 = p1;
         p3 = str2;
         while(*p2++ == *p3++)
         {
            if(*p3 == '\0')
               return(0);
         }
      }
      p1++;
   }
   return(1);
}
```

```
/*****************************************************************
* This function is used to search a string for a pattern
* and replace the pattern with a new pattern.
*****************************************************************/

replace(line,pat,new_pat)
char *line, *pat, *new_pat;

{
   char *p1, *p2, *p3;
   char *save_p1;
   char buf[256];
   int i, got_pat;

   if(match(line,pat) != 0)   /* pattern is not in the line */
      return;
   p1 = line;
   got_pat = 0;
   i = 0;
   while(*p1)
   {
      if(*p1 == *pat && !got_pat)
      {
         p2 = p1;
         p3 = pat;
         while((*p2++ == *p3++) && !got_pat)
         {
            if(*p3 == '\0')
            {
               p1 = p2;
               while(*new_pat)
                  buf[i++] = *new_pat++;
               got_pat = 1;
               break;
            }
         }
         buf[i++] = *p1++;
      }
      else
         buf[i++] = *p1++;
   }
   buf[i++] = '\0';
   strcpy(line,buf);
   return;
}
```

```
/*****************************************************************
 Reset the tty states to their original values and exit.
*****************************************************************/
onintrpt()
{
    signal(SIGINT,SIG_IGN);
    if((access(exec_paths,0)) == 0)
        unlink(exec_paths);
    mvcur(0,COLS-1,LINES-1,0);
    endwin();
    exit(0);
}
```

```
/*********************************************************
*
*
* install.c - Install test cases in the file system
*
* This file contains the functions ilang, icomp, and iclas.
* These functions handle the installation of the argument.
* The date and the user id of the person installing the
* test cases is placed in the "class_log" file associated
* with the installed class.
*
* The following global variables are set in these functions:
*   langpth - current language directory
*   comp_pth - current compiler directory
*   classpth1 - current class directory at the language level
*
* If an error is detected, a "1" is
* returned; otherwise a "0" is returned.
*
**********************************************************/

#include "arts.h"
#include "header.h"
#include <pwd.h>

struct stat stbuf;

/*********************************************************
*
* This function handles installation of all languages
* The language pointer points to the language string
**********************************************************/

ilang(language)
char *language;
{
   FILE *fptr;
   char buf[LINESZ];
   char fpath[LINESZ];
   char line[LINESZ];

   lower(language);   /* convert all languages to lower case */
   strcpy(buf,language);
   convert(buf);   /* convert language name to directory name */
   sprintf(langpth,"%s/%s",root,buf);
```

```
if(stat(root, &stbuf) == -1)   /* directory does not exist */
{
    signal(SIGINT,SIG_IGN);   /* Ignore interrupts */
    sprintf(buf,"mkdir %s",root);
    system(buf);
    signal(SIGINT,onintrpt);
}
sprintf(fpath,"%s/LANGUAGES",root);
if((fptr = fopen(fpath,"r"))==NULL) /* no language file exists */
{
    signal(SIGINT,SIG_IGN);   /* Ignore interrupts */
    fptr = fopen(fpath,"w");
    fprintf(fptr,"%s\n",language);
    sprintf(buf,"mkdir %s",langpth);
    system(buf);
    fclose(fptr);
    signal(SIGINT,onintrpt);
    return(0);
}
else /* languages file exists */
{
    while((fgets(line,LINESZ,fptr)) != NULL)
    {
        line[strlen(line) - 1] = '\0';   /* get rid of NL */
        if(strcmp(line,language) == 0)   /* language is in db */
        {
            fclose(fptr);
            return(0);
        }
    }
}
fclose(fptr);
if(stat(langpth, &stbuf) == -1)   /* directory does not exist */
{
    signal(SIGINT,SIG_IGN);   /* Ignore interrupts */
    sprintf(buf,"mkdir %s",langpth);
    system(buf);
    signal(SIGINT,onintrpt);
}
else
{
    errormsg("Language name conflict. Choose another name");
    return(0);
}
signal(SIGINT,SIG_IGN);   /* Ignore interrupts */
fptr = fopen(fpath,"a");
```

```
    fprintf(fptr,"%s\n",language);
    fclose(fptr);
    signal(SIGINT,onintrpt);
    return(0);
}

/**********************************************************
*
* This function handles installation of all compilers
* The compiler pointer points to the compiler string
**********************************************************/

icomp(compiler)
char *compiler;
{
    FILE *fptr;
    char buf[LINESZ];
    char fpath[LINESZ];
    char line[LINESZ];

    lower(compiler);  /* convert all languages to lower case */
    strcpy(buf,compiler);
    convert(buf);  /* convert compiler name to directory name */

    sprintf(comp_pth,"%s/%s",langpth,buf);
    sprintf(fpath,"%s/COMPILERS",langpth);
    if((fptr = fopen(fpath,"r"))==NULL) /* no compiler file exists */
    {
        signal(SIGINT,SIG_IGN);  /* Ignore interrupts */
        fptr = fopen(fpath,"w");
        fprintf(fptr,"%s\n",compiler);
        sprintf(buf,"mkdir %s",comp_pth);
        system(buf);
        fclose(fptr);
        signal(SIGINT,onintrpt);
        return(0);
    }
    else /* COMPILERS file exists */
    {
        while((fgets(line,LINESZ,fptr)) != NULL)
        {
            line[strlen(line) - 1] = '\0';  /* get rid of NL */
            if(strcmp(line,compiler) == 0)  /* compiler is in db */
            {
                fclose(fptr);
                return(0);
            }
        }
    }
    fclose(fptr);
    if(stat(comp_pth, &stbuf) == -1)   /* directory does not exist */
    {
        signal(SIGINT,SIG_IGN);   /* Ignore interrupts */
        sprintf(buf,"mkdir %s",comp_pth);
        system(buf);
        signal(SIGINT,onintrpt);
    }
    else
    {
        errormsg("Compiler name conflict. Choose another name");
        return(0);
    }
    signal(SIGINT,SIG_IGN);   /* Ignore interrupts */
    fptr = fopen(fpath,"a");
    fprintf(fptr,"%s\n",compiler);
    fclose(fptr);
    signal(SIGINT,onintrpt);
    return(0);
}

/************************************************************
*
* This function handles installation of all classes
* The class pointer points to the class string
*************************************************************/

iclas(class)
char *class;
{
    FILE *fptr;
    char buf[LINESZ];
    char fpath[LINESZ];
    char line[LINESZ];

    lower(class);   /* convert all classs to lower case */
    strcpy(buf,class);
    convert(buf);   /* convert class name to directory name */
    sprintf(classpth1,"%s/%s",langpth,buf);

    sprintf(fpath,"%s/CLASSES",langpth);
    if((fptr = fopen(fpath,"r"))==NULL) /* no class file exists */
    {
        signal(SIGINT,SIG_IGN);   /* Ignore interrupts */
        fptr = fopen(fpath,"w");
        fprintf(fptr,"%s\n",class);
        fclose(fptr);
        sprintf(buf,"mkdir %s",classpth1);
        system(buf);
        signal(SIGINT,onintrpt);
        return(0);
    }
    else /* classs file exists */
    {
        while((fgets(line,LINESZ,fptr)) != NULL)
        {
            line[strlen(line) - 1] = '\0';   /* get rid of NL */
            if(strcmp(line,class) == 0)   /* class is in db */
            {
                fclose(fptr);
                return(0);
            }
        }
    }
    fclose(fptr);
    if(stat(classpth1, &stbuf) == -1)   /* directory does not exist */
    {
        signal(SIGINT,SIG_IGN);   /* Ignore interrupts */
        sprintf(buf,"mkdir %s",classpth1);
        system(buf);
        signal(SIGINT,onintrpt);
    }
    else
    {
        errormsg("Class name conflict. Choose another name");
        return(0);
    }
    signal(SIGINT,SIG_IGN);   /* Ignore interrupts */
    fptr = fopen(fpath,"a");
    fprintf(fptr,"%s\n",class);
    fclose(fptr);
    signal(SIGINT,onintrpt);
    return(0);
}
/************************************************************
*
* This function handles installation of the class_log info.
* The class_log info is the person installing the test cases
* and the date.
* The class pointer points to the class string
**************************************************************/
ihead()
{
   long time();
   char *ctime();
   FILE *fptr;
   char buf[LINESZ];
   char *getlogin();
   char *name, *date;
   long time();
   long *clock;

   name = getlogin();  /* get the login id of the user */
   sprintf(buf,"%s/class_log",classpth1);
   signal(SIGINT,SIG_IGN);  /* Ignore interrupts */
   fptr = fopen(buf,"a");

fprintf(fptr,"********************************************\n ");
   fprintf(fptr,"Installed by: %s\n",name);
   clock = time((long *) 0);
   date = ctime(&clock);
   fprintf(fptr,"%s\n",date);

fprintf(fptr,"********************************************\n ");
   fclose(fptr);
   signal(SIGINT,onintrpt);
   return(0);
}
```

```
#include "arts.h"
#include "header.h"

/************************************************************
*
*
* installw.c - ARTS installation window.
*
* flg is passed as a parameter to this function.
*    flg = 0 then standard installation
*    flg = 1 then install exec_spec only
*
*************************************************************/

installw(flg)
int flg;

{
   FILE *fptr;

   int row, r;
   int got_compiler, got_test_cases;
   char resp[LINESZ];
   char buf[256];
   char line[LINESZ];
   char file[LINESZ];
   char path[LINESZ];
   char compiler[15];
   char suffix[15];
   char t_cases[15];

   bldinstalwin();  /* build the installation window */
   cbreak();
   nl();
   noecho();
   wclear(mainwin);
   wrefresh(mainwin);
   row = 7;

   for(;;)  /* get the language */
   {
      wmove(instalwin,row,2);
      waddstr(instalwin,"Enter language: ");
      wclrtoeol(instalwin);
      box(instalwin,'*','*');
      wrefresh(instalwin);
```

```
    mywgetstr(instalwin,resp);
    clearerror();
    if(resp[0] == 'q' && resp[1] == '\0')   /* return to main menu */
        return;
    else if(resp[0] == '?' && resp[1] == '\0')   /* list languages */
    {
        sprintf(buf,"%s/LANGUAGES",root);
        if((fptr = fopen(buf,"r")) == NULL)
        {
            statusmsg("There are no languages installed");
            continue;
        }
        overwrite(instalwin,mainwin);
        wclear(instalwin);
        wmove(instalwin,1,5);
        waddstr(instalwin,"Installed languages:");
        r = 3;
        while((fgets(buf,LINESZ,fptr)) != NULL)
        {
            wmove(instalwin,r++,1);
            waddstr(instalwin,buf);
        }
        fclose(fptr);
        r++;
        wmove(instalwin,r++,5);
        waddstr(instalwin,"Press any key to continue");
        wrefresh(instalwin);
        wgetch(instalwin);
        overwrite(mainwin,instalwin);
        wclear(mainwin);
        wrefresh(instalwin);
        continue;
    }
    else if(resp[0] == '\0')   /* RETURN was entered */
    {
        errormsg("A language or 'q' must be entered");
        continue;
    }
    if(flg && (glang(resp) != 0))   /* get the language */
        continue;   /* A failure occurred */
    else if((ilang(resp)) != 0)   /* install the language */
        continue;   /* A failure occurred */
    row++;
    break;
}
```

```
for(;;)  /* install the compiler */
{
    wmove(instalwin,row,2);
    waddstr(instalwin,"Enter compiler: ");
    wclrtoeol(instalwin);
    box(instalwin,'*','*');
    wrefresh(instalwin);
    mywgetstr(instalwin,resp);
    clearerror();
    if(resp[0] == 'q' && resp[1] == '\0') /* return to main menu */
        return;
    else if(resp[0] == '?' && resp[1] == '\0') /* list compilers */
    {
        sprintf(buf,"%s/COMPILERS",langpth);
        if((fptr = fopen(buf,"r")) == NULL)
        {
            statusmsg("There are no compilers for this language installed");
            continue;
        }
        overwrite(instalwin,mainwin);
        wclear(instalwin);
        wmove(instalwin,1,5);
        waddstr(instalwin,"Installed compilers:");
        r = 3;
        while((fgets(buf,LINESZ,fptr)) != NULL)
        {
            wmove(instalwin,r++,1);
            waddstr(instalwin,buf);
        }
        fclose(fptr);
        r++;
        wmove(instalwin,r++,5);
        waddstr(instalwin,"Press any key to continue");
        wrefresh(instalwin);
        wgetch(instalwin);
        overwrite(mainwin,instalwin);
        wclear(mainwin);
        wrefresh(instalwin);
        continue;
    }
    else if(resp[0] == '\0')   /* RETURN was entered */
    {
        errormsg("A compiler or 'q' must be entered");
        continue;
    }
    if(flg && (gcomp(resp) != 0))   /* get the compiler */
```

```
            continue;   /* A failure occurred */
        else if(icomp(resp) != 0)   /* install the compiler */
            continue;   /* A failure occurred */
        strcpy(compiler,resp);
        lower(compiler);   /* convert the compiler to lower case */
        row++;
        break;
    }


/**********************************************************
    * We must now check the compiler_info file for the compiler
    * If it is not there, we must prompt for a path to the
    * compiler and append the compiler and path in the file.
**********************************************************/
    got_compiler = 0;
    sprintf(buf,"%s/COMPILER_INFO",root);
    if((fptr = fopen(buf,"r")) != NULL)
    {
        while(fgets(line,LINESZ,fptr) != NULL)
        {
            if(strcmp(compiler,(strtok(line,":"))) == 0)
            {
                strtok(NULL,":");   /* don't need the path */
                strcpy(suffix,(strtok(NULL,"\n")));
                got_compiler = 1;
                break;
            }
        }
        fclose(fptr);
    }
    if(!got_compiler && !flg)  /* compiler was not found */
    {
        for(;;)   /* get the path to the compiler */
        {
            wmove(instalwin,row,2);
            waddstr(instalwin,"Enter path to compiler: ");
            wclrtoeol(instalwin);
            box(instalwin,'*','*');
            wrefresh(instalwin);
            mywgetstr(instalwin,resp);
            clearerror();
            if(resp[0] == 'q' && resp[1] == '\0')
                return;
            else if(resp[0] == '?' && resp[1] == '\0')
            {
```

```
            statusmsg("Enter the full UNIX path to the compiler");
            continue;
        }
        else if(resp[0] == '\0')
        {
            errormsg("A path or 'q' must be entered");
            continue;
        }
        else if((access(resp,0)) != 0)
        {
            sprintf(buf,"Cannot access %s",resp);
            errormsg(buf);
            continue;
        }
        strcpy(path,resp);
        row++;
        break;
    }
}
if(!got_compiler && !flg)
{
    for(;;)  /* get the suffix required by the compiler */
    {
        wmove(instalwin,row,2);
        waddstr(instalwin,"Test case file suffix (if any): ");
        wclrtoeol(instalwin);
        box(instalwin,'*','*');
        wrefresh(instalwin);
        mywgetstr(instalwin,resp);
        clearerror();
        if(resp[0] == 'q' && resp[1] == '\0')
            return;
        else if(resp[0] == '?' && resp[1] == '\0')
        {
            statusmsg("Enter any suffix \(eg. .c\) which may be needed");
            continue;
        }
        strcpy(suffix,resp);
        break;
    }
    sprintf(buf,"%s/COMPILER_INFO",root);
    signal(SIGINT,SIG_IGN);
    fptr = fopen(buf,"a");
    fprintf(fptr,"%s:%s:%s\n",compiler,path,suffix);
    fclose(fptr);
    signal(SIGINT,onintrpt);
    row++;
}

sprintf(t_cases,"test_cases%s",suffix);

for(;;)  /* get the class */
{
    wmove(instalwin,row,2);
    waddstr(instalwin,"Enter class: ");
    wclrtoeol(instalwin);
    box(instalwin,'*','*');
    wrefresh(instalwin);
    mywgetstr(instalwin,resp);
    clearerror();
    if(resp[0] == 'q' && resp[1] == '\0')  /* return to main menu */
        return;
    else if(resp[0] == '?' && resp[1] == '\0')   /* list classes */
    {
        sprintf(buf,"%s/CLASSES",langpth);
        if((fptr = fopen(buf,"r")) == NULL)
        {
            statusmsg("There are no classes for this language installed");
            continue;
        }
        overwrite(instalwin,mainwin);
        wclear(instalwin);
        wmove(instalwin,1,5);
        waddstr(instalwin,"Installed classes:");
        r = 3;
        while((fgets(buf,LINESZ,fptr)) != NULL)
        {
            wmove(instalwin,r++,1);
            waddstr(instalwin,buf);
        }
        fclose(fptr);
        r++;
        wmove(instalwin,r++,5);
        waddstr(instalwin,"Press any key to continue");
        wrefresh(instalwin);
        wgetch(instalwin);
        overwrite(mainwin,instalwin);
        wclear(mainwin);
        wrefresh(instalwin);
        continue;
    }
    else if(resp[0] == '\0')  /* RETURN was entered */
```

```
        {
            errormsg("A class or 'q' must be entered");
            continue;
        }
        if((iclas(resp)) != 0)   /* install the class */
            continue;   /* A failure occurred */
        row++;
        break;
    }

    for(;;)   /* get the test case file */
    {
        if(flg)   /* exec_spec only is being installed */
            break;
        wmove(instalwin,row,2);
        waddstr(instalwin,"Test case file: ");
        wclrtoeol(instalwin);
        box(instalwin,'*','*');
        wrefresh(instalwin);
        mywgetstr(instalwin,resp);
        clearerror();
        if(resp[0] == 'q' && resp[1] == '\0')   /* return to main menu */
            return;
        else if(resp[0] == '?' && resp[1] == '\0')
        {
            statusmsg("Enter name of file which contain your test cases");
            continue;
        }
        else if(resp[0] == '\0')
        {
            errormsg("A test case file or 'q' must be entered");
            continue;
        }
        if((fptr = fopen(resp,"r")) == NULL)
        {
            sprintf(buf,"Cannot read %s",resp);
            errormsg(buf);
            continue;
        }
        fclose(fptr);
        signal(SIGINT,SIG_IGN);

        /* copy the test case file into the ARTS file system */
        sprintf(buf,"cp %s %s/%s",resp,classpth1,t_cases);
        system(buf);
        signal(SIGINT,onintrpt);
        strcpy(file,resp);
        row++;
        break;
    }

    for(;;)   /* get any input data */
    {
        if(flg)   /* exec_spec only is being installed */
            break;
        wmove(instalwin,row,2);
        waddstr(instalwin,"Input data file (if any): ");
        wclrtoeol(instalwin);
        box(instalwin,'*','*');
        wrefresh(instalwin);
        mywgetstr(instalwin,resp);
        clearerror();
        if(resp[0] == 'q' && resp[1] == '\0')
            return;
        else if(resp[0] == '?' && resp[1] == '\0')
        {
            statusmsg("Enter name of file which contain your input data");
            continue;
        }
        else if(resp[0] == '\0')
            break;
        if((fptr = fopen(resp,"r")) == NULL)
        {
            sprintf(buf,"Cannot read %s",resp);
            errormsg(buf);
            continue;
        }
        fclose(fptr);
        signal(SIGINT,SIG_IGN);

        /* copy the input data file into the ARTS file system */
        sprintf(buf,"cp %s %s/input_data",resp,classpth1);
        system(buf);
        signal(SIGINT,onintrpt);
        row++;
        break;
    }

    if(!flg)
    {
        if((ihead()) != 0)   /* install the class_log */
            return;   /* A failure occurred */
        sprintf(buf,"The file %s has been installed",file);
        statusmsg(buf);
    }

    /* Build the execution specification file */
    mktemp(tmpfile);
    fptr = fopen(tmpfile,"w");
    wclear(instalwin);
    mvwaddstr(instalwin,1,2,"Enter the Execution Script");
    mvwaddstr(instalwin,3,2,"Use $COMPILER for the Compiler or");
    mvwaddstr(instalwin,4,2,"Use $INTERPRETER for an Interpreter");
    mvwaddstr(instalwin,5,2,"Enter a '.' as the first character on a line");
    mvwaddstr(instalwin,6,2,"and press RETURN to enter the script");
    mvwaddstr(instalwin,7,2,"Enter a 'q' as the first character on a line");
    mvwaddstr(instalwin,8,2,"and press RETURN to quit and return to the
         Main Menu");
    r = 10;   /* row counter */
    got_compiler = 0;   /* flag to show that a keyword was entered */
    got_test_cases = 0; /* flag to show that test_case file was entered */
    for(;;)
    {
        box(instalwin,'*','*');
        wmove(instalwin,r++,2);
        waddstr(instalwin,"==> ");
        wrefresh(instalwin);
        mywgetstr(instalwin,buf);
        clearerror();
        clearstat();
        if(buf[0] == 'q' && buf[1] == '\0')   /* return to main menu */
        {
            fclose(fptr);
            unlink(tmpfile);
            wclear(instalwin);
            wrefresh(instalwin);
            return;
        }
        if(buf[0] == '?' && buf[1] == '\0')
        {
            statusmsg("Enter a line which is needed by the execution
specification");
            continue;
        }
        if(buf[0] == '.' && buf[1] == '\0')   /* end it */
        {
            /* both compiler and test cases file must be specified */
            if(got_compiler && got_test_cases)
    {
        fclose(fptr);
        signal(SIGINT,SIG_IGN);
        sprintf(buf,"mv %s %s/exec_spec",tmpfile,classpth1);
        system(buf);
        signal(SIGINT,onintrpt);
        statusmsg("The exec_spec file has been installed");
        wclear(instalwin);
        wrefresh(instalwin);
        return;
    }
    /* let's try again */
    else if(!got_compiler)
        sprintf(buf,"$COMPILER or $INTERPRETER not entered");
    else
        sprintf(buf,"'%s' has not been entered",t_cases);
    errormsg(buf);
    rewind(fptr);
    r = 10;
    got_compiler = got_test_cases = 0;
    wmove(instalwin,r,5);
    wclrtobot(instalwin);
    wrefresh(instalwin);
    continue;
}
if(r == 20)   /* allow only 10 lines */
{
    got_compiler = got_test_cases = 0;
    errormsg("Too many lines");
    rewind(fptr);
    r = 10;
    wmove(instalwin,r,5);
    wclrtobot(instalwin);
    wrefresh(instalwin);
    continue;
}
if(match(buf,t_cases) == 0 )  /* check for test_cases */
    got_test_cases = 1;

/* Check for $COMPILER and $INTERPRETER */
if((match(buf,"$COMPILER") == 0) && (match(buf,"$INTERPRETER") == 0))
{
    errormsg("Only one $COMPILER or $INTERPRETER may be entered");
    if(match(buf,t_cases) == 0 )
        got_test_cases = 0;
    wmove(instalwin,r,5);
```

```
            wclrtoeol(instalwin);
            --r;
            continue;
        }

        /* Check for $COMPILER or $INTERPRETER */
        if((match(buf,"$COMPILER") == 0) || (match(buf,"$INTERPRETER") == 0))
        {
            if(got_compiler)
            {
                errormsg("Only one $COMPILER or $INTERPRETER may be entered");
                if(match(buf,t_cases) == 0 )
                    got_test_cases = 0;
                wmove(instalwin,r,5);
                wclrtoeol(instalwin);
                --r;
                continue;
            }
            got_compiler = 1;
        }
        fprintf(fptr,"%s\n",buf);
    }
}
```

```
#include "arts.h"
#include "header.h"

/************************************************************
*
* mainw.c - ARTS main window
*
* Controls the curser movement in the main window.
* When a selection is made by the user, the
* appropriate function is called. An exit is
* made from ARTS by selecting the EXIT function
* or by hitting the BREAK key.
*
************************************************************/

mainw()
{
   int resp;
   int row, col;
   char buf[LINESZ];

   cbreak();
   nonl();
   noecho();
   row = 10;
   col = COLS - 64;
   bldmainwin();   /* build the main window */

   for(;;)
   {
      wmove(mainwin,row,col);
      wrefresh(mainwin);
      resp = wgetch(mainwin);
      if(resp == '   ' || resp == '\n') /* move to the next row */
      {
         if(row == (19))   /* reset the row counter */
            row = 10;
         else
            row = row + 1;  /* increment the row counter */
         continue;
      }
      else if(resp == 'x' || resp == 'X')   /* a selection was made */
      {
         waddch(mainwin,resp);
         wrefresh(mainwin);
         clearerror();
```

```
clearstat();
switch (row) {
case 10:   /* Install test cases */
    installw(0);
    wmove(instalwin,0,0);
    wclrtobot(instalwin);
    wrefresh(instalwin);
    break;
case 11:   /* Select classes for execution */
    selectw();
    wmove(selectwin,0,0);
    wclrtobot(selectwin);
    wrefresh(selectwin);
    break;
case 12:   /* Begin execution */
    execute();
    break;
case 13:   /* Install exec_spec */
    installw(1);
    wclear(instalwin);
    wrefresh(instalwin);
    break;
case 14:   /* print testing analysis */
    sprintf(buf,"%s/rje/test_analysis",home);
    more(buf);
    wclear(scrolwin);
    wrefresh(scrolwin);
    break;
case 15:   /* generate a report of an execution */
    report();
    break;
case 16:   /* retrieve class data files */
    retrievw();
    wclear(retrievewin);
    wrefresh(retrievewin);
    break;
case 17:   /* delete class */
    deletew();
    wclear(deletewin);
    wrefresh(deletewin);
    break;
case 18:   /* print helpfile */
    sprintf(buf,"%s/helpfile",root);
    more(buf);
    wclear(scrolwin);
    /* wrefresh(scrolwin); */
```

```
                break;
            case 19:   /* exit */
                if((access(exec_paths,0)) == 0)
                    unlink(exec_paths);
                mvcur(0,COLS-1,23,0);
                wclrtoeol(errorwin);
                endwin();
                exit(0);
            }
            bldmainwin();
            cbreak();
            nonl();
            noecho();
            row = 10;   /* reset the row counter */
        }
    }
}
```

```
/************************************************************
**
*
* more.c - a Curses version of a pagination routine
*
*     A pointer to the file to be displayed is required
*     as input to this function.
*
*************************************************************/

#include "arts.h"
#include "header.h"

more(ptr)
char *ptr;  /* this is the file to be displayed */
{
   char buf[LINESZ];
   char line[256];
   FILE *rptr;
   int row;

   if((rptr = fopen(ptr,"r")) == NULL)
   {
      sprintf(line,"Cannot open %s",ptr);
      errormsg(line);
      return;
   }
   wclear(mainwin);
   wrefresh(mainwin);
   idlok(scrolwin,TRUE);
   for(;;)
   {
      wmove(scrolwin,0,0);
      for(row=0;row < 20;row++)   /* scroll for 20 lines*/
      {
         if(fgets(line,256,rptr) == NULL)
         {
            fclose(rptr);
            wmove(scrolwin,20,0);
            waddstr(scrolwin,"RETURN: ");
            wrefresh(scrolwin);
            wgetch(scrolwin);
            wclear(scrolwin);
            wrefresh(scrolwin);
            idlok(scrolwin,FALSE);
            return;
            }
            line[strlen(line) -1] = '\0';
            wmove(scrolwin,row,0);
            waddstr(scrolwin,line);
        }
        wmove(scrolwin,20,0);
        waddstr(scrolwin,"more (q to quit): ");
        wrefresh(scrolwin);
        if(wgetch(scrolwin) == 'q')
        {
            fclose(rptr);
            wclear(scrolwin);
            wrefresh(scrolwin);
            idlok(scrolwin,FALSE);
            return;
        }
        wclear(scrolwin);
        wrefresh(scrolwin);
    }
}
```

```
/********************************************************
*
* post_exec.c - ARTS post execution processor
*
* Create a "test_analysis" file in the user's rje directory.
* Place the following information in this file:
*    user's id
*    date and time of test run
*    language
*    compiler under test
*    compiler standard
*    classes which were tested
*    PASS or FAIL
*    output from a "diff" of the testing results for
*      each class.
*
* Create an "analysis" file for each class of the compiler
* under test and place the output of the "diff" operation in
* this file.
*
* Append the following information to the "class_log" file for
* each class being tested:
*    user's id
*    date and time of test run
*    compiler under test
*    compiler standard
*    PASS or FAIL
*
* Return a "0" if no errors were detected; otherwise, return
* a "1".
*
********************************************************/

#include "arts.h"
#include "header.h"

post_exec()
{

    FILE *rptr;
    FILE *fptr;
    char line[256];
    char buf[LINESZ];
    char class[LINESZ];
    char c_u_tst[LINESZ];    /* path to comp_under_tst dir */
    char c_std[LINESZ];      /* path to comp_std dir */
    char class_log[LINESZ];     /* class_log file for the class being tested */
    char status[8];    /* PASSED or FAILED */
    long *time();
    char *ctime();
    char *strrchr();
    char *getlogin();
    char *name, *date;
    long *clock;

    if((rptr = fopen(exec_paths,"r")) == NULL)
    {
        sprintf(line,"Cannot read %s",exec_paths);
        errormsg(line);
        return(1);
    }
    name = getlogin();   /* get the login id of the user */
    clock = time((long *) 0);
    date = ctime(&clock);  /* get the current date and time */
    sprintf(buf,"%s/rje/test_analysis",home);
    fptr = fopen(buf,"w");

fprintf(fptr,"*************************************************************\n");
    fprintf(fptr,"\n\t\t\tARTS TESTING RESULTS\n\n");
    fprintf(fptr,"Testing performed by: %s\n",name);
    fprintf(fptr,"%s\n\n",date);

fprintf(fptr,"*************************************************************\n\n");
    fclose(fptr);
    while((fgets(line,256,rptr)) != NULL)
    {
        if(strchr(line,':') == NULL)
        {
            errormsg("Improper exec_paths format");
            fclose(rptr);
            return(1);
        }
        strcpy(class,strtok(line,":"));
        strcpy(class_log,strtok(NULL,":"));
        strcpy(c_u_tst,strtok(NULL,":"));
        strcpy(c_std,strtok(NULL,"\n"));
        sprintf(buf,"%s/output_data",c_u_tst);
        if((access(buf,0)) != 0)   /* compiler under test output */
        {
            sprintf(line,"Cannot access %s,.continuing",buf);
            errormsg(line);
            continue;
        }
        sprintf(buf,"%s/output_data",c_std);
        if((access(buf,0)) != 0)   /* compiler standard output */
        {
            sprintf(line,"Cannot access %s, continuing",buf);
            errormsg(line);
            continue;
        }

        /* do a "diff" of the two output files */
        sprintf(line,"diff %s/output_data %s/output_data > %s/analysis",
         c_u_tst,c_std,c_u_tst);
        system(line);
        sprintf(buf,"%s/analysis",c_u_tst);
        if((fptr = fopen(buf,"r")) == NULL)
        {
            sprintf(line,"Cannot open %s",buf);
            errormsg(buf);
            continue;
        }
        if(fgets(line,256,fptr) == NULL)
            strcpy(status,"PASSED");
        else
            strcpy(status,"FAILED");
        fclose(fptr);

        update_head(class_log,status,name,date,c_u_tst,c_std);
        update_results(status,c_u_tst,c_std); /* update analysis */
    }
    statusmsg("Completed analysis of data");
    fclose(rptr);
    return(0);
}

/******************************************************************
* This function places the id of the person performing the test run,
* the current date, the compilers being tested, and the results
* of the test run in the class_log file.
*******************************************************************/
update_head(hpth,status,name,date,pth1,pth2)
char *hpth, *status, *name, *date, *pth1, *pth2;
{
    FILE *fptr;
    char buf[LINESZ];

    sprintf(buf,"%s/class_log",hpth);
    signal(SIGINT,SIG_IGN);
    fptr = fopen(buf,"a");

fprintf(fptr,"********************************************\n");
    fprintf(fptr,"Test run performed by: %s\n",name);
    fprintf(fptr,"%s",date);
    fprintf(fptr,"Compiler under test: %s\n",pth1);
    fprintf(fptr,"Compiler standard: %s\n",pth2);
    fprintf(fptr,"Status - %s\n",status);

fprintf(fptr,"********************************************\n");
    fclose(fptr);
    signal(SIGINT,onintrpt);
    return;
}

/*******************************************************************
* This function places the compilers being tested, and the results
* of the test run in the analysis file in the user's rje directory.
* If the test failed, the diff output is appended to the file.
* pth1 = compiler under test
* pth2 = compiler standard
*******************************************************************/
update_results(status,pth1,pth2)
char *status, *pth1, *pth2;
{
    FILE *fptr;
    char buf[256];
    int i;
    char *ptr1, *ptr2;
    char cls[15], comp1[15], comp2[15], lang[15];
    char c_u_tst[LINESZ];

    strcpy(c_u_tst,pth1);
    sprintf(buf,"%s/rje/test_analysis",home);
    fptr = fopen(buf,"a");


fprintf(fptr,"********************************************\n");

    /* Get the lang, class, and comp_under_tst */
    ptr1 = strrchr(pth1,'/');
    strcpy(cls,ptr1+1);
    *ptr1 = '\0';
    ptr2 = strrchr(pth1,'/');
    strcpy(comp1,ptr2+1);
    *ptr2 = '\0';
    ptr1 = strrchr(pth1,'/');
    strcpy(lang,ptr1+1);

    /* Get the compiler standard */
    ptr1 = strrchr(pth2,'/');
    *ptr1 = '\0';
    ptr2 = strrchr(pth2,'/');
    strcpy(comp2,ptr2+1);

    fprintf(fptr,"Language: %s\n",lang);
    fprintf(fptr,"Compiler under test: %s\n",comp1);
    fprintf(fptr,"Compiler standard: %s\n",comp2);
    fprintf(fptr,"Class: %s\n",cls);
    fprintf(fptr,"Status of tests for this class: %s\n",status);
    fclose(fptr);
    if(strcmp(status,"FAILED") == 0)
    {
        sprintf(buf,"cat %s/analysis >> %s/rje/test_analysis",c_u_tst,home);
        system(buf);
    }
}
```

```
/*****************************************************************
*
* report.c - Generate a report of the results of a test run.
*
* A "test_analysis" file must be in the user's rje directory.
*
*****************************************************************/
#include "arts.h"
#include "header.h"

report()
{
   char buf[LINESZ];

   if((home = (char *) getenv("HOME")) == NULL)
   {
      errormsg("The HOME variable is not set");
      return;
   }
   sprintf(buf,"%s %s/rje/test_analysis",PRINT_CMD,home);
   system(buf);
   statusmsg("The report has been sent to the line printer");
}
```

```
/************************************************************
*
*
* retrieve.c - Build paths to various parts of the file system
*
* The following global variables are set in these functions:
*   langpth - current language directory
*   comp_pth - current compiler directory
*   classpth1 - current class directory at the language level
*   classpth2 - current class directory at the compiler level
*
* This module contains the functions glang, gcomp, and gclas.
* If an error is detected, a "1" is
* returned; otherwise a "0" is returned.
*
*************************************************************/

#include "arts.h"
#include "header.h"

/************************************************************
*
* This function sets the langpth to the language directory.
* The language pointer points to the language string
*************************************************************/

glang(language)
char *language;
{
   FILE *fptr;
   char buf[LINESZ];
   char fpath[LINESZ];
   char line[LINESZ];

   lower(language);  /* convert all languages to lower case */
   strcpy(buf,language);
   convert(buf);  /* convert language name to directory name */

   sprintf(fpath,"%s/LANGUAGES",root);
   if((fptr = fopen(fpath,"r"))==NULL) /* no language file exists */
   {
      errormsg("No languages installed");
      return(1);
   }
   else /* languages file exists */
   {
        while((fgets(line,LINESZ,fptr)) != NULL)
        {
            line[strlen(line) - 1] = '\0';   /* get rid of NL */
            if(strcmp(line,language) == 0)   /* language is in db */
            {
                sprintf(langpth,"%s/%s",root,buf);
                fclose(fptr);
                return(0);
            }
        }
    }
    fclose(fptr);
    sprintf(buf,"The language %s is not installed",language);
    errormsg(buf);
    return(1);
}

/**********************************************************
*
* This function handles retrieval of all compilers
* The compiler pointer points to the compiler string
**********************************************************/

gcomp(compiler)
char *compiler;
{
    FILE *fptr;
    char buf[LINESZ];
    char fpath[LINESZ];
    char line[LINESZ];

    lower(compiler);   /* convert all languages to lower case */
    strcpy(buf,compiler);
    convert(buf);   /* convert compiler name to directory name */

    sprintf(fpath,"%s/COMPILERS",langpth);
    if((fptr = fopen(fpath,"r"))==NULL) /* no compiler file exists */
    {
        errormsg("No compilers or interpreters installed for the language");
        return(1);
    }
    else /* compilers file exists */
    {
        while((fgets(line,LINESZ,fptr)) != NULL)
        {
            line[strlen(line) - 1] = '\0';   /* get rid of NL */
        if(strcmp(line,compiler) == 0)   /* compiler is in db */
        {
           sprintf(comp_pth,"%s/%s",langpth,buf);
           fclose(fptr);
           return(0);
        }
     }
  }
  fclose(fptr);
  sprintf(buf,"The compiler %s is not installed",compiler);
  errormsg(buf);
  return(1);
}

/************************************************************
*
* This function handles retrieval of all classes
* The class pointer points to the class string
*************************************************************/

gclas(class)
char *class;
{
  FILE *fptr;
  char buf[LINESZ];
  char fpath[LINESZ];
  char line[LINESZ];

  lower(class);   /* convert all classs to lower case */
  strcpy(buf,class);
  convert(buf);   /* convert class name to directory name */

  sprintf(fpath,"%s/CLASSES",langpth);
  if((fptr = fopen(fpath,"r"))==NULL) /* no class file exists */
  {
     errormsg("No classes exist for the compiler");
     return(1);
  }
  else /* classs file exists */
  {
     while((fgets(line,LINESZ,fptr)) != NULL)
     {
        line[strlen(line) - 1] = '\0';   /* get rid of NL */
        if(strcmp(line,class) == 0)   /* class is in db */
        {
           fclose(fptr);
```

```
                sprintf(classpth1,"%s/%s",langpth,buf);
                sprintf(classpth2,"%s/%s",comp_pth,buf);
                return(0);
            }
        }
    }
    fclose(fptr);
    sprintf(buf,"The class %s is not installed",class);
    errormsg(buf);
    return(1);
}
```

```
#include "arts.h"
#include "header.h"

/**********************************************************
*
*
* retrievw.c - ARTS test case retrieval window
*
* All files located in the class directories
* are copied into the user's rje directory.
* These files include: test_cases, class_log,
* output_data, exec_spec, analysis, and
* input_data.
*
***********************************************************/

retrievw()
{
   char *getenv();
   FILE *fptr;
   char resp[LINESZ];
   char buf[LINESZ];
   char file[LINESZ];
   int row;

   bldretrievewin();  /* build the retrieve window */
   cbreak();
   nl();
   noecho();
   wclear(mainwin);
   wrefresh(mainwin);

   for(;;)  /* get the language */
   {
      wmove(retrievewin,7,2);
      waddstr(retrievewin,"Enter language: ");
      wclrtoeol(retrievewin);
      box(retrievewin,'*','*');
      wrefresh(retrievewin);
      mywgetstr(retrievewin,resp);
      clearerror();
      if(resp[0] == 'q' && resp[1] == '\0')  /* return to main menu */
         return;
      else if(resp[0] == '?' && resp[1] == '\0') /* list languages */
      {
         sprintf(buf,"%s/languages",root);
```

```
        if((fptr = fopen(buf,"r")) == NULL)
        {
           statusmsg("There are no languages installed");
           continue;
        }
        overwrite(retrievewin,deletewin);
        wclear(retrievewin);
        wmove(retrievewin,1,5);
        waddstr(retrievewin,"Installed languages:");
        row = 3;
        while((fgets(buf,LINESZ,fptr)) != NULL)
        {
           wmove(retrievewin,row++,1);
           waddstr(retrievewin,buf);
        }
        fclose(fptr);
        row++;
        wmove(retrievewin,row++,5);
        waddstr(retrievewin,"Press any key to continue");
        wrefresh(retrievewin);
        wgetch(retrievewin);
        wclear(retrievewin);
        overwrite(deletewin,retrievewin);
        wclear(deletewin);
        wrefresh(retrievewin);
        continue;
     }
     else if(resp[0] == '\0')  /* RETURN was entered */
     {
        errormsg("A language or 'q' must be entered");
        continue;
     }
     else if((glang(resp)) != 0)   /* get the language */
        continue;   /* A failure occurred */
     break;
  }

  for(;;)   /* get the compiler */
  {
     wmove(retrievewin,8,2);
     waddstr(retrievewin,"Enter compiler: ");
     wclrtoeol(retrievewin);
     box(retrievewin,'*','*');
     wrefresh(retrievewin);
     mywgetstr(retrievewin,resp);
     clearerror();
```

```
    if(resp[0] == 'q' && resp[1] == '\0')  /* return to main menu */
        return;
    else if(resp[0] == '?' && resp[1] == '\0')   /* list compilers */
    {
        sprintf(buf,"%s/compilers",langpth);
        if((fptr = fopen(buf,"r")) == NULL)
        {
            statusmsg("There are no compilers for the language installed");
            continue;
        }
        overwrite(retrievewin,deletewin);
        wclear(retrievewin);
        wmove(retrievewin,1,5);
        waddstr(retrievewin,"Installed compilers:");
        row = 3;
        while((fgets(buf,LINESZ,fptr)) != NULL)
        {
            wmove(retrievewin,row++,1);
            waddstr(retrievewin,buf);
        }
        fclose(fptr);
        row++;
        wmove(retrievewin,row++,5);
        waddstr(retrievewin,"Press any key to continue");
        wrefresh(retrievewin);
        wgetch(retrievewin);
        wclear(retrievewin);
        overwrite(deletewin,retrievewin);
        wclear(deletewin);
        wrefresh(retrievewin);
        continue;
    }
    else if(resp[0] == '\0')   /* RETURN was entered */
    {
        errormsg("A compiler or 'q' must be entered");
        continue;
    }
    else if((gcomp(resp)) != 0)   /* get the compiler */
        continue;   /* A failure occurred */
    break;
}

for(;;)   /* get the class */
{
    wmove(retrievewin,9,2);
    waddstr(retrievewin,"Enter class: ");
```

```
        wclrtoeol(retrievewin);
        box(retrievewin,'*','*');
        wrefresh(retrievewin);
        mywgetstr(retrievewin,resp);
        clearerror();
        if(resp[0] == 'q' && resp[1] == '\0')   /* return to main menu */
            return;
        else if(resp[0] == '?' && resp[1] == '\0')   /* list classes */
        {
            sprintf(buf,"%s/CLASSES",langpth);
            if((fptr = fopen(buf,"r")) == NULL)
            {
                statusmsg("There are no classes for the language installed");
                continue;
            }
            overwrite(retrievewin,deletewin);
            wclear(retrievewin);
            wmove(retrievewin,1,5);
            waddstr(retrievewin,"Installed classes:");
            row = 3;
            while((fgets(buf,LINESZ,fptr)) != NULL)
            {
                wmove(retrievewin,row++,1);
                waddstr(retrievewin,buf);
            }
            fclose(fptr);
            row++;
            wmove(retrievewin,row++,5);
            waddstr(retrievewin,"Press any key to continue");
            wrefresh(retrievewin);
            wgetch(retrievewin);
            wclear(retrievewin);
            overwrite(deletewin,retrievewin);
            wclear(deletewin);
            wrefresh(retrievewin);
            continue;
        }
        else if(resp[0] == '\0')   /* a RETURN was entered */
        {
            errormsg("A class or 'q' must be entered");
            continue;
        }
        else if((gclas(resp)) != 0)   /* get the class */
            continue;   /* A failure occurred */
        break;
    }
    sprintf(buf,"cp %s/* %s/rje",classpth1,home);
    system(buf);
    sprintf(buf,"cp %s/* %s/rje",classpth2,home);
    system(buf);
    statusmsg("All data files have been copied to your rje directory");
}
```

```
/***********************************************************
*
* select.c - Select the test cases for execution
*
* This module contains the functions slang, scomp, sclas,
* and endselect.
* A character string is expected as a parameter for all
* functions, except endselect. In addition, a 1 or 2 is
* expected as an additional parameter for the scomp
* function. 1 signals the compiler under test, 2 signals
* the compiler standard.
*
* The following global variables are set in these functions:
*   langpth - current language directory
*   comp_pth - current compiler directory for compiler under test
*   comp_pth2 - current compiler directory for the compiler standard
*   classpth1 - current class directory at the language level
*   classpth2 - current class directory at the compiler level
*
* If an error is detected, a "1" is
* returned; otherwise a "0" is returned.
*
***********************************************************/

#include "arts.h"
#include "header.h"

/***********************************************************
*
* This function handles retrieval of two compilers.
* The compiler pointer points to the compiler string.
* The select integer is a flag to the type of compiler selected.
*    A "1" is for the compiler under test.
*    A "2" is for the compiler standard.
***********************************************************/

scomp(compiler,select)
char *compiler;
int select;
{
   FILE *fptr;
   char buf[LINESZ];
   char fpath[LINESZ];
   char line[LINESZ];
   char dirname[LINESZ];
   lower(compiler);   /* convert all languages to lower case */
   strcpy(dirname,compiler);
   convert(dirname);   /* convert compiler name to directory name */

   sprintf(fpath,"%s/COMPILERS",langpth);
   if((fptr = fopen(fpath,"r"))==NULL) /* no compiler file exists */
   {
      errormsg("No compilers or interpreters installed for the language");
      return(1);
   }
   else /* COMPILERS file exists */
   {
      while((fgets(line,LINESZ,fptr)) != NULL)
      {
         line[strlen(line) - 1] = '\0';   /* get rid of NL */
         if(strcmp(line,compiler) == 0)   /* compiler is in db */
         {
            if(select == 1)   /* compiler under test */
               sprintf(comp_pth,"%s/%s",langpth,dirname);
            else /* compiler standard */
               sprintf(comp_pth2,"%s/%s",langpth,dirname);
            fclose(fptr);
            return(0);
         }
      }
   }
   fclose(fptr);
   sprintf(buf,"The compiler %s is not installed",compiler);
   errormsg(buf);
   return(1);
}

/************************************************************
*
* This function handles retrieval of all classes.
* The class pointer points to the class string.
************************************************************/

sclas(class,c_undr_tst,c_std)
char *class;   /* class being tested */
char *c_undr_tst;   /* compiler under test */
char *c_std;   /* compiler standard */
{
   FILE *rptr;
   FILE *wptr1;
   FILE *wptr2;
```

```
char buf[LINESZ];
char fpath[LINESZ];
char line[LINESZ];
char pth1[LINESZ];
char pth2[LINESZ];
char pth3[LINESZ];
char dirname[LINESZ];
int got_class;

lower(class);   /* convert all classes to lower case */
strcpy(dirname,class);
convert(dirname);   /* convert class name to directory name */

if((wptr1 = fopen(exec_paths,"a")) == NULL)
{
   sprintf(buf,"Cannot open %s",exec_paths);
   errormsg(buf);
   return(1);
}
sprintf(fpath,"%s/CLASSES",langpth);
got_class = 0;
if((rptr = fopen(fpath,"r"))==NULL) /* no class file exists */
{
   errormsg("No classes exist for the compiler");
   fclose(wptr1);
   return(1);
}
else /* class file exists */
{
   while((fgets(line,LINESZ,rptr)) != NULL)
   {
      line[strlen(line) - 1] = '\0';   /* get rid of NL */
      if(strcmp(line,class) == 0)   /* class is in db */
      {
         fclose(rptr);
         got_class = 1;
         sprintf(pth1,"%s/%s",langpth,dirname);
         sprintf(pth2,"%s/%s",comp_pth,dirname);
         sprintf(pth3,"%s/%s",comp_pth2,dirname);
         break;
      }
   }
}
if(!got_class)
{
   fclose(rptr);
```

```
        fclose(wptr1);
        sprintf(buf,"The class %s is not installed",class);
        errormsg(buf);
        return(1);
    }


/****************************************************************
* pth1 = dir path to the exec_spec files
* pth2 = dir path to the comp_under_test output files
* pth3 = dir path to the compiler standard output files
****************************************************************/
    fprintf(wptr1,"%s:%s:%s:%s\n",class,pth1,pth2,pth3);
    fclose(wptr1);
    sprintf(fpath,"%s/exec_spec",pth1);
    if((rptr = fopen(fpath,"r"))==NULL) /* no class file exists */
    {
        errormsg("The exec_spec file has not been built for this class");
        return(1);
    }
    sprintf(buf,"%s/.exec_spec1",pth1);
    wptr1 = fopen(buf,"w");   /* this is for the compiler under test */
    sprintf(buf,"%s/.exec_spec2",pth1);
    wptr2 = fopen(buf,"w");   /* this is for the compiler standard */
    while((fgets(line,LINESZ,rptr)) != NULL)
    {
        strcpy(buf,line);
        if(match(line,"$COMPILER") == 0)
        {
            replace(buf,"$COMPILER",c_undr_tst);
            replace(line,"$COMPILER",c_std);
        }
        else if(match(line,"$INTERPRETER") == 0)
        {
            replace(buf,"$INTERPRETER",c_undr_tst);
            replace(line,"$INTERPRETER",c_std);
        }
        fprintf(wptr1,"%s",buf);
        fprintf(wptr2,"%s",line);
    }

    fclose(rptr);
    fclose(wptr1);
    fclose(wptr2);
    return(0);
}
```

```
#include "arts.h"
#include "header.h"

/************************************************************
*
*
* selectw.c - ARTS test case selection window
*
* The language, compiler under test, compiler standard and
* one or more classes for a test run are selected.
*
*************************************************************/
selectw()
{
   FILE *fptr;

   int row, r;
   char buf[256];
   char line[256];
   char resp[LINESZ];
   char comp[LINESZ];
   char c_undr_tst[LINESZ];   /* path to compiler under test */
   char c_std[LINESZ];   /* path to compiler standard */

   bldselectwin();   /* build the selection window */
   cbreak();
   nl();
   noecho();
   wclear(mainwin);
   wrefresh(mainwin);

   for(;;)   /* select a language */
   {
      wmove(selectwin,7,2);
      waddstr(selectwin,"Enter language: ");
      wclrtoeol(selectwin);
      box(selectwin,'*','*');
      wrefresh(selectwin);
      mywgetstr(selectwin,resp);
      clearerror();
      if(resp[0] == 'q' && resp[1] == '\0')   /* return to main menu */
         return;
      else if(resp[0] == '?' && resp[1] == '\0') /* list languages */
      {
         sprintf(buf,"%s/LANGUAGES",root);
         if((fptr = fopen(buf,"r")) == NULL)
```

```
        {
            statusmsg("There are no languages installed");
            continue;
        }
        overwrite(selectwin,mainwin);
        wclear(selectwin);
        wmove(selectwin,1,5);
        waddstr(selectwin,"Installed languages:");
        row = 3;
        while((fgets(buf,LINESZ,fptr)) != NULL)
        {
            wmove(selectwin,row++,1);
            waddstr(selectwin,buf);
        }
        fclose(fptr);
        row++;
        wmove(selectwin,row++,5);
        waddstr(selectwin,"Press any key to continue");
        wrefresh(selectwin);
        wgetch(selectwin);
        wclear(selectwin);
        overwrite(mainwin,selectwin);
        wclear(mainwin);
        wrefresh(selectwin);
        continue;
    }
    else if(resp[0] == '\0')   /* RETURN was entered */
    {
        errormsg("A language or 'q' must be entered");
        continue;
    }
    else if((glang(resp)) != 0)   /* select the language */
        continue;   /* A failure occurred */
    break;
}

for(;;)   /* select the compiler under test */
{
    wmove(selectwin,8,2);
    waddstr(selectwin,"Enter compiler under test: ");
    wclrtoeol(selectwin);
    box(selectwin,'*','*');
    wrefresh(selectwin);
    mywgetstr(selectwin,resp);
    clearerror();
    if(resp[0] == 'q' && resp[1] == '\0')   /* return to main menu */
```

```
            return;
        else if(resp[0] == '?' && resp[1] == '\0')   /* list compilers */
        {
            sprintf(buf,"%s/COMPILERS",langpth);
            if((fptr = fopen(buf,"r")) == NULL)
            {
                statusmsg("There are no compilers installed");
                continue;
            }
            overwrite(selectwin,mainwin);
            wclear(selectwin);
            wmove(selectwin,1,5);
            waddstr(selectwin,"Installed compilers:");
            row = 3;
            while((fgets(buf,LINESZ,fptr)) != NULL)
            {
                wmove(selectwin,row++,1);
                waddstr(selectwin,buf);
            }
            fclose(fptr);
            row++;
            wmove(selectwin,row++,5);
            waddstr(selectwin,"Press any key to continue");
            wrefresh(selectwin);
            wgetch(selectwin);
            overwrite(mainwin,selectwin);
            wclear(mainwin);
            wrefresh(selectwin);
            continue;
        }
        else if(resp[0] == '\0')
        {
            errormsg("A compiler or 'q' must be entered");
            continue;
        }
        else if((scomp(resp,1)) != 0)   /* select the compiler under test */
            continue;   /* A failure occurred */
        strcpy(comp,resp);
        break;
    }

    /* Try to get the path out of the compiler info file */

    sprintf(buf,"%s/COMPILER_INFO",root);
    if((fptr = fopen(buf,"r")) != NULL)
    {
```

```
    while((fgets(line,256,fptr)) != NULL)
    {
        if(strcmp(comp,strtok(line,":")) == 0)
        {
            strcpy(c_undr_tst,strtok(NULL,":"));
            break;
        }
    }
    fclose(fptr);
}

wmove(selectwin,9,2);
sprintf(buf,"Path to compiler: %s",c_undr_tst);
waddstr(selectwin,buf);

for(;;)   /* select the compiler standard */
{
    wmove(selectwin,10,2);
    waddstr(selectwin,"Enter standard compiler: ");
    wclrtoeol(selectwin);
    box(selectwin,'*','*');
    wrefresh(selectwin);
    mywgetstr(selectwin,resp);
    lower(resp);   /* Change to lower case */
    clearerror();
    if(resp[0] == 'q' && resp[1] == '\0')   /* return to main menu */
        return;
    else if(resp[0] == '?' && resp[1] == '\0')  /* list compilers */
    {
        sprintf(buf,"%s/COMPILERS",langpth);
        if((fptr = fopen(buf,"r")) == NULL)
        {
            statusmsg("There are no compilers installed");
            continue;
        }
        overwrite(selectwin,mainwin);
        wclear(selectwin);
        wmove(selectwin,1,5);
        waddstr(selectwin,"Installed compilers:");
        row = 3;
        while((fgets(buf,LINESZ,fptr)) != NULL)
        {
            wmove(selectwin,row++,1);
            waddstr(selectwin,buf);
        }
        fclose(fptr);
        row++;
        wmove(selectwin,row++,5);
        waddstr(selectwin,"Press any key to continue");
        wrefresh(selectwin);
        wgetch(selectwin);
        overwrite(mainwin,selectwin);
        wclear(mainwin);
        wrefresh(selectwin);
        continue;
    }
    else if(resp[0] == '\0')
    {
        errormsg("A compiler or 'q' must be entered");
        continue;
    }
    else if((strcmp(resp,comp)) == 0)
    {
        errormsg("Standard compiler cannot be the same as the
    compiler under test");
        continue;
    }
    else if((scomp(resp,2)) != 0)   /* select the compiler standard */
        continue;   /* A failure occurred */
    break;
}

/* Try to get the path out of the compiler info file */
sprintf(buf,"%s/COMPILER_INFO",root);
if((fptr = fopen(buf,"r")) != NULL)
{
    while((fgets(line,256,fptr)) != NULL)
    {
        if(strcmp(resp,strtok(line,":")) == 0)
        {
            strcpy(c_std,strtok(NULL,":"));
            break;
        }
    }
    fclose(fptr);
}

wmove(selectwin,11,2);
sprintf(buf,"Path to compiler: %s",c_std);
waddstr(selectwin,buf);

row = 12;   /* initialize the row counter */
```

```
for(;;)  /* select one or more classes */
{
   if(row == 19)  /* re-initialize the window for more classes */
   {
      row = 1;
      wclear(selectwin);
      box(selectwin,'*','*');
      wrefresh(selectwin);
   }
   wmove(selectwin,row,2);
   waddstr(selectwin,"Enter class ('all' or '.' to end): ");
   wclrtoeol(selectwin);
   box(selectwin,'*','*');
   wrefresh(selectwin);
   mywgetstr(selectwin,resp);
   clearerror();
   if(resp[0] == 'q' && resp[1] == '\0')   /* return to main menu */
      return;
   else if(strcmp(resp,"all") == 0)   /* all classes was selected */
   {
      sprintf(buf,"%s/CLASSES",langpth);
      if((fptr = fopen(buf,"r")) == NULL)
      {
         errormsg("There are no classes installed");
         continue;
      }
      statusmsg("Selecting all classes");
      while((fgets(buf,LINESZ,fptr)) != NULL)
      {
         buf[strlen(buf) - 1] = '\0';   /* get rid of NL */
         sclas(buf,c_undr_tst,c_std);  /* select the class */
      }
      fclose(fptr);
      break;
   }
   else if(resp[0] == '?' && resp[1] == '\0')   /* list classes */
   {
      sprintf(buf,"%s/CLASSES",langpth);
      if((fptr = fopen(buf,"r")) == NULL)
      {
         statusmsg("There are no classes installed");
         continue;
      }
      overwrite(selectwin,mainwin);
      wclear(selectwin);
      wmove(selectwin,1,5);
```

```
            waddstr(selectwin,"Installed classes:");
            r = 3;
            while((fgets(buf,LINESZ,fptr)) != NULL)
            {
                wmove(selectwin,r++,1);
                waddstr(selectwin,buf);
            }
            fclose(fptr);
            r++;
            wmove(selectwin,r++,5);
            waddstr(selectwin,"Press any key to continue");
            wrefresh(selectwin);
            wgetch(selectwin);
            overwrite(mainwin,selectwin);
            wrefresh(selectwin);
            wclear(mainwin);
            continue;
        }
        else if(resp[0] == '\0')   /* RETURN was entered */
        {
            errormsg("A class, 'q', or '.' must be entered");
            continue;
        }
        else if(resp[0] == '.' && resp[1] == '\0')   /* end selections */
            break;
        else if((sclas(resp,c_undr_tst,c_std)) != 0)   /* select the class */
            continue;
        ++row;
    }
}
```

APPENDIX C

Sample Prolog Test Cases

```
/*************************************************************
* These test cases test the following Comparison of Terms
* built-in predicates:
*       ==
*       ==
*       @<
*       @>
*       @=<
*       @>=
*
* This set of test cases can be run with the following
* command:
*    Cprolog <<!
*    ['test_cases.pl'].
*    test.
*    !
*
*************************************************************/

test :- test(1,94).
test(I,N) :- I>N,!.
test(I,N) :- I=<N, (dotest(I);true),!,J is I+1, test(J,N).

w(X) :- write(X).

dotest(1) :- w('Testing "==" using equal integers: '),
       (-17 == -17 ->
       w('passed') ;
       w('FAILED')),
       nl.
dotest(2) :- w('Testing "==" for failure using unequal integers: '),
       (0 == -17 ->
       w('FAILED') ;
       w('passed')),
       nl.
dotest(3) :- w('Testing "==" using equal atoms: '),
       (xyz == xyz ->
       w('passed') ;
       w('FAILED')),
       nl.
dotest(4) :- w('Testing "==" for failure using unequal atoms: '),
       (xyy == xyz ->
       w('FAILED') ;
```

```
        w('passed')),
        nl.
dotest(5) :- w('Testing "==" using equal complex terms: '),
        (compare(=,1,1) == compare(=,1,1) ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(6) :- w('Testing "==" for failure using complex terms of different names: '),
        (compare(=,1,1) == cmpare(=,1,1) ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(7) :- w('Testing "==" for failure using unequal functor arguments: '),
        (compare(=,1,2) == compare(=,1,1) ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(8) :- w('Testing "==" for failure using complex terms of different arity: '),
        (re(=,1,1) == re(1,1) ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(9) :- w('Testing "==" using variables instantiated to equal values: '),
        X=17,Y=17,
        (X == Y ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(10) :- w('Testing "==" for failure using variables
instantiated to unequal values: '),
        X=17,Y= -17,
        (X == Y ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(11) :- w('Testing "==" for failure using an atom and an integer: '),
        (xyz == 17 ->
        w('FAILED') ;
        w('passed')),
        nl.

dotest(12) :- w('Testing "==" using unequal integers: '),
        (17 == -17 ->
        w('passed') ;
        w('FAILED')),
        nl.
```

```
dotest(13) :- w('Testing "==" for failure using equal integers: '),
        (-17 == -17 ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(14) :- w('Testing "==" using unequal atoms: '),
        (xyz == xyy ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(15) :- w('Testing "==" for failure using equal atoms: '),
        (xyz == xyz ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(16) :- w('Testing "==" using unequal functor arguments: '),
        (compare(=,1,2) == compare(=,1,1) ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(17) :- w('Testing "==" using complex terms of different arity: '),
        (re(=,1,1) == re(1,1) ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(18) :- w('Testing "==" using complex terms of different names: '),
        (ree(1,1) == re(1,1) ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(19) :- w('Testing "==" for failure using equal complex terms: '),
        (compare(=,1,1) == compare(=,1,1) ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(20) :- w('Testing "==" using variables instantiated to unequal values: '),
        X=7,Y=17,
        (X == Y ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(21) :- w('Testing "==" for failure using variables
instantiated to equal values: '),
        X=17,Y=17,
        (X == Y ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(22) :- w('Testing "==" using an atom and an integer: '),
        (xyz == 17 ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(23) :- w('Testing "@<" using unequal integers: '),
        (17 @< 18 ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(24) :- w('Testing "@<" for failure using equal integers: '),
        (17 @< 17 ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(25) :- w('Testing "@<" for failure using unequal integers: '),
        (0 @< -7 ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(26) :- w('Testing "@<" using unequal atoms: '),
        (xyy @< xyz ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(27) :- w('Testing "@<" for failure using equal atoms: '),
        (xyz @< xyz ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(28) :- w('Testing "@<" for failure using unequal atoms: '),
        (xyz @< xyy ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(29) :- w('Testing "@<" using complex terms of different names: '),
        (compar(=,1,1) @< compare(=,1,1) ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(30) :- w('Testing "@<" for failure using equal complex terms: '),
        (compare(=,1,1) @< compare(=,1,1) ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(31) :- w('Testing "@<" for failure using unequal functor arguments: '),
        (compare(=,1,2) @< compare(=,1,1) ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(32) :- w('Testing "@<" for failure using complex terms of different arity: '),
        (re(=,1,1) @< re(1,1) ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(33) :- w('Testing "@<" using variables instantiated to unequal values: '),
        X = -6,Y = 17,
        (X @< Y ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(34) :- w('Testing "@<" for failure using variables
instantiated to equal values: '),
        X = -6,Y = -6,
        (X @< Y ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(35) :- w('Testing "@<" for failure using variables
instantiated to unequal values: '),
        X = -6,Y = -7,
        (X @< Y ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(36) :- w('Testing "@<" using an atom and an integer: '),
        (17 @< xyz ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(37) :- w('Testing "@<" for failure using an atom and an integer: '),
        (xyz @< 17 ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(38) :- w('Testing "@>" using unequal integers: '),
        (18 @> 17 ->
        w('passed') ;
        w('FAILED')),
        nl.
```

```
dotest(39) :- w('Testing "@>" for failure using equal integers: '),
        (17 @> 17 ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(40) :- w('Testing "@>" for failure using unequal integers: '),
        (-33 @> 0 ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(41) :- w('Testing "@>" using unequal atoms: '),
        (xyz @> xyy ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(42) :- w('Testing "@>" for failure using equal atoms: '),
        (xyz @> xyz ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(43) :- w('Testing "@>" for failure using unequal atoms: '),
        (xyy @> xyz ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(47) :- w('Testing "@>" using unequal complex terms: '),
        (compare(=,1,1) @> compar(=,1,1) ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(48) :- w('Testing "@>" for failure using equal complex terms: '),
        (compare(=,1,1) @> compare(=,1,1) ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(49) :- w('Testing "@>" for failure using unequal complex terms: '),
        (compar(=,1,1) @> compare(=,1,1) ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(50) :- w('Testing "@>" using variables instantiated to unequal values: '),
        X = -6,Y = 17,
        (Y @> X ->
        w('passed') ;
        w('FAILED')),
        nl.
```

```
dotest(51) :- w('Testing "@>" for failure using variables
instantiated to equal values: '),
        X = -6,Y = -6,
        (Y @> X ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(52) :- w('Testing "@>" for failure using variables
instantiated to unequal values: '),
        X = -7,Y = -6,
        (X @> Y ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(53) :- w('Testing "@>" using unequal functor arguments: '),
        (compare(=,1,2) @>= compare(=,1,1) ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(54) :- w('Testing "@>" for failure using unequal functor arguments: '),
        (compare(=,1,2) @>= compare(=,2,1) ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(55) :- w('Testing "@>" using complex terms of different arity: '),
        (re(=,1,1) @>= re(1,1) ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(56) :- w('Testing "@>" for failure using complex terms of different arity: '),
        (re(1,1) @> re(=,1,1) ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(57) :- w('Testing "@>" using an atom and an integer: '),
        (xyz @> 17 ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(58) :- w('Testing "@>" for failure using an atom and an integer: '),
        (17 @> xyz ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(59) :- w('Testing "@=<" using unequal integers: '), (17 @=< 18 ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(60) :- w('Testing "@=<" using equal integers: '),
        (17 @=< 17 ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(61) :- w('Testing "@=<" for failure using unequal integers: '),
        (0 @=< -7 ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(62) :- w('Testing "@=<" using unequal atoms: '),
        (xyy @=< xyz ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(63) :- w('Testing "@=<" using equal atoms: '),
        (xyz @=< xyz ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(64) :- w('Testing "@=<" for failure using unequal atoms: '),
        (xyz @=< xyy ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(65) :- w('Testing "@=<" using complex terms of different names: '),
        (compar(=,1,1) @=< compare(=,1,1) ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(66) :- w('Testing "@=<" using equal complex terms: '),
        (compare(=,1,1) @=< compare(=,1,1) ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(67) :- w('Testing "@=<" for failure using unequal functor arguments: '),
        (compare(=,1,2) @=< compare(=,1,1) ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(68) :- w('Testing "@=<" for failure using complex terms of different arity: '),
        (re(=,1,1) @=< re(1,1) ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(69) :- w('Testing "@=<" using variables instantiated to unequal values: '),
        X = -6,Y = 17,
        (X @=< Y ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(70) :- w('Testing "@=<" using variables instantiated to equal values: '),
        X = -6,Y = -6,
        (X @=< Y ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(71) :- w('Testing "@=<" for failure using variables
instantiated to unequal values: '),
        X = -6,Y = -7,
        (X @=< Y ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(72) :- w('Testing "@=<" using an atom and an integer: '),
        (17 @=< xyz ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(73) :- w('Testing "@=<" for failure using an atom and an integer: '),
        (xyz @=< 17 ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(74) :- w('Testing "@>=" using unequal integers: '),
        (18 @>= 17 ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(75) :- w('Testing "@>=" using equal integers: '),
        (17 @>= 17 ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(76) :- w('Testing "@>=" for failure using unequal integers: '),
        (-33 @>= 0 ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(77) :- w('Testing "@>=" using unequal atoms: '),
        (xyz @>= xyy ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(78) :- w('Testing "@>=" using equal atoms: '),
        (xyz @>= xyy ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(79) :- w('Testing "@>=" for failure using unequal atoms: '),
        (xyy @>= xyz ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(83) :- w('Testing "@>=" using unequal complex terms: '),
        (compare(=,1,1) @>= compar(=,1,1) ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(84) :- w('Testing "@>=" using equal complex terms: '),
        (compare(=,1,1) @>= compare(=,1,1) ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(85) :- w('Testing "@>=" for failure using unequal complex terms: '),
        (compar(=,1,1) @>= compare(=,1,1) ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(86) :- w('Testing "@>=" using variables instantiated to unequal values: '),
        X = -6,Y = 17,
        (Y @>= X ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(87) :- w('Testing "@>=" using variables instantiated to equal values: '),
        X = -6,Y = -6,
        (Y @>= X ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(88) :- w('Testing "@>=" for failure using variables
instantiated to unequal values: '),
        X = -7,Y = -6,
        (X @>= Y ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(89) :- w('Testing "@>=" using unequal functor arguments: '),
        (compare(=,1,2) @>= compare(=,1,1) ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(90) :- w('Testing "@>=" for failure using unequal functor arguments: '),
        (compare(=,1,2) @>= compare(=,2,1) ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(91) :- w('Testing "@>=" using complex terms of different arity: '),
        (re(=,1,1) @>= re(1,1) ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(92) :- w('Testing "@>=" for failure using complex terms of different arity: '),
        (re(1,1) @>= re(=,1,1) ->
        w('FAILED') ;
        w('passed')),
        nl.
dotest(93) :- w('Testing "@>=" using an atom and an integer: '),
        (xyz @>= 17 ->
        w('passed') ;
        w('FAILED')),
        nl.
dotest(94) :- w('Testing "@>=" for failure using an atom and an integer: '),
        (17 @>= xyz ->
        w('FAILED') ;
        w('passed')),
        nl.
```

An Automated Regression Testing System for Compilers and Interpreters

by

Robert L. Horney

B.S., Metropolitan 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

1988

ABSTRACT

This paper presents a survey of current tools and techniques which are used for the dynamic validation of software. Also presented in this paper is the requirements and design for the Automated Regression Test System (ARTS).

Early error detection and correction is important in determining the success or failure of a software product. Several studies have shown us that it is extremely important to detect software errors as early as possible in a product's life cycle. Many software tools and techniques have been developed to aid us in the detection of errors or "bugs". The common goal of these tools and techniques is to demonstrate that the product being tested is free of errors.

Several tools which have been developed for the specific purpose of validating the correctness of compilers are examined. These validation systems can be classified as being either test case generators or test case drivers. Archiving systems and comparators may also be part of the compiler validation tool.

The Automated Regression Test System presented in this paper is a test case driver and archiver which can be used to regression test compilers or interpreters which run on the UNIX† operating system. Although the ARTS is limited to regression testing only, it has the flexibility required to regression test any compiler or interpreter supported by UNIX.

† UNIX is a trademark of Bell Laboratories