

AD-A127 333

PRELIMINARY DESIGN AND IMPLEMENTATION OF A METHOD FOR
VALIDATING EVOLVING ADA COMPILERS(U) AIR FORCE INST OF
TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI..

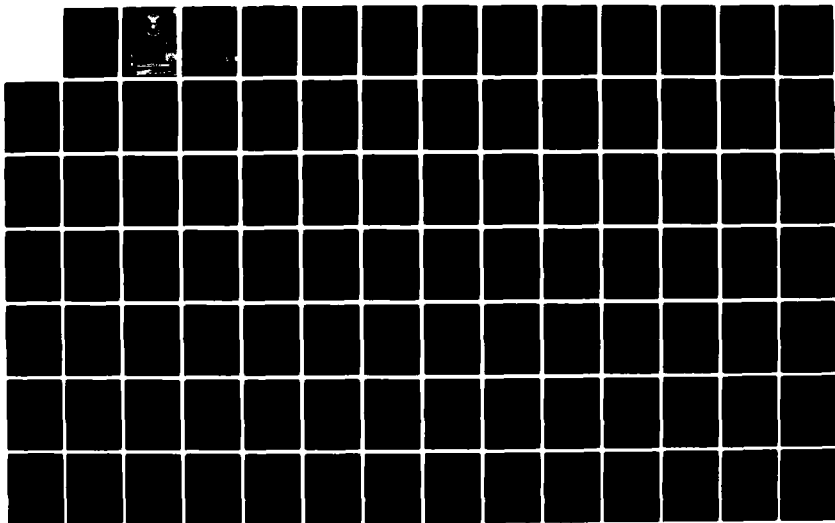
1/2

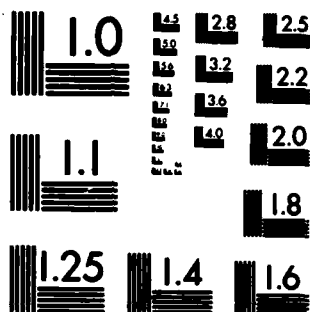
UNCLASSIFIED

E D MILLER MAR 83 AFIT/GCS/MA/83M-1

F/G 9/2

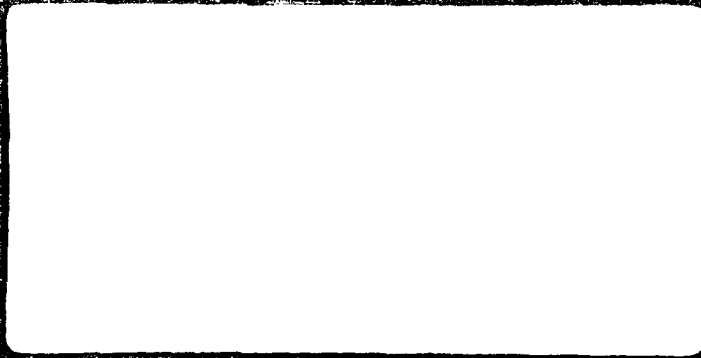
NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A127333



AFIT/GCS/MA/83M-1 ✓

PRELIMINARY DESIGN AND IMPLEMENTATION
OF A METHOD FOR VALIDATING EVOLVING
ADA COMPILERS

THESIS

AFIT/GCS/MA/83M-1

Edward D. Miller
Capt USA

DTIC
ELECTE
S APR 28 1983 D
E

Approved for public release; distribution unlimited.

AFIT/GCS/MA/83M-1

PRELIMINARY DESIGN AND IMPLEMENTATION
OF A METHOD FOR VALIDATING EVOLVING
ADA COMPILERS

THESIS

Presented to the faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

by
Edward D. Miller Jr.
Capt USA
Graduate Computer Science
March 1983

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A	



Approved for public release; distribution unlimited.

PREFACE

The Department of Defense funded the development of the Ada Compiler Validation Capability (ACVC) for use in the validation-testing of Ada compilers. The ACVC was targeted to test only those compilers which implement the entire Ada language. It appeared that the development of a testing capability for evolving Ada compilers would be a very useful tool for compiler developers. The need for such a tool coupled with my desire to learn more about Ada led to the selection of this thesis topic.

I would like to thank my advisor, Captain Roie R. Black, for all the time and guidance he has given me. His ideas and suggestions during the course of the project were most helpful. I would also like to thank my thesis-committee members, Lieutenant Colonel Harold W. Carter and Major Michael R. Varrieur. Their suggestions and comments were very valuable.

Deep gratitude is also expressed to Patricia A. Knoop of the Language Control Branch, Computer Operations Division, Aeronautical Systems Division, Wright-Patterson AFB, Ohio, who originally proposed the topic and provided the resources needed during the project.

Finally, I would like to thank my parents for all the support and encouragement they have given me.

Contents

	Page
1. INTRODUCTION	1
1.1 Background -- DoD's Software Problem	1
1.1.1 The proliferation of languages	2
1.1.2 The High Order Language Working Group	3
1.1.3 The search for a solution	3
1.1.4 Design Phase	4
1.1.5 The need for a standard	4
1.1.6 The Ada Compiler Validation Capability	5
1.1.7 Ada compiler development	6
1.1.8 The need for an incremental validation capability	6
1.2 Objective	7
1.3 General Approach	7
1.4 Sequence of Presentation	8
2. ANALYSIS	10
2.1 Introduction	10
2.2 Background	10
2.2.1 Approach to testing	11
2.2.2 Test set design	11
2.2.3 Test set organization	13
2.3 General structure	17
2.3.1 Report Package	17
2.3.2 Tests	18
2.4 Detailed Analysis	19
2.4.1 Class A	20
2.4.2 Class C	24
2.4.3 Class D	26
2.4.4 Class B	27
2.4.5 Class L	28
2.4.6 Class E	29
2.5 Summary	29
3. PROJECT DEVELOPMENT	30
3.1 Introduction	30
3.2 Overview	30
3.3 Identification of language-features	32
3.4 Development of a representation	35
3.5 Removal of unsupported features	39
3.5.1 Outputting the modified representation	44
3.6 Summary	44
4. IMPLEMENTATION	45
4.1 Introduction	45
4.2 Input	45
4.3 Data Structures	46
4.4 Output	48
4.5 Limitations	48

Contents

	Page
5. Recommendations and Conclusions	53
5.1 Introduction	53
5.2 Recommendations	53
5.3 Conclusions	54
Bibliography	56
Appendix A - Evolving Compiler Development	57
Appendix B - Report Package	60
Appendix C - BNF	66
Appendix D - Removal of language-features from valid tests	85
Appendix E - Manual evaluation of tests	89
Appendix F - Description of Garlington Compiler	100
Appendix G - Modifications to the Garlington compiler	102
Appendix H - Empty productions	103
Appendix I - Recursive productions	104

List of Figures

<u>Figure</u>		<u>Page</u>
2-1	LRM definition of identifier structure	15
2-2	ACVC test objective and design guidelines	16
2-3	Class A Test	21
2-4	Repackaged Structure for Class A tests	22
2-5	Class C test	24
2-6	Restructured Class C test	25
2-7	Class D test	26
2-8	Class B test	27
2-9	Class L test	28
3-1	Overview of Test Development Process	31
3-2	Internal structure of Modify Test Program	32
3-3	Parse tree representation	36
3-4	Representation of a production	36
3-5	Representation of Program ::= compl_unit_list	38
3-6	Representation of Production 4 is added	38
3-7	Representation of a recursive production	40
3-8	Representation of a potentially empty production	42
3-9	Representation after production is eliminated	43
4-1	Data structure used to represent productions	47
4-2	Test with records and arrays removed	50
4-3	Test with record removed	51
4-4	Complete test	52
D-1	Class A test	86
D-2	Class A test with record removed	87
D-3	Class A test with records and arrays removed	88
E-1	Test segment analyzed manually	89

ABSTRACT



This project consisted of a preliminary design and a partial implementation of a tool which modifies the existing Ada Compiler Validation Capability (ACVC) test set so it can be used to test evolving Ada compilers. The project evaluated the feasibility of repackaging each of test classes found in the ACVC and suggested methods for repackaging the tests. The tool developed uses a table-driven parser which parses the July 1980 proposed standard. It uses output from the parser to generate a representation of a test program. Once the representation is developed, unsupported language-features are removed from it. The remaining representation is output as a valid test program.



1. INTRODUCTION

In order to combat the rising cost of software in embedded computer systems, the Department of Defense (DoD) sponsored efforts which led to the design of the Ada programming language. The efforts were not limited to the design of a new programming language; DoD also sponsored efforts to develop the Ada Programming Support Environment (APSE) and the Ada Compiler Validation Capability (ACVC).

This thesis project is based on the results of the ACVC developed by "The Software Technology Company" (SofTech). Specifically, it investigates the problem of modifying tests contained in the ACVC so they can be used to test evolving Ada compilers (described in Appendix A). The project was proposed and sponsored by the Language Control Branch, Computer Operations Division, Aeronautical Systems Division Computer Center, Wright-Patterson AFB.

This chapter begins by providing background information on the development of the Ada programming language and the ACVC. It concludes with an introductory description of the thesis project.

1.1 Background -- DoD's Software Problem

Intensive studies completed in the early 1970's identified the software costs associated with embedded computer systems as the most significant DoD software

problem. These studies also revealed that the majority of the embedded computer system costs were related to software maintenance rather than software development (Ref 4:24).

1.1.1 The proliferation of languages

A principle factor contributing to the rising cost of embedded computer system software was the large number of languages used within the DoD. Over 450 general-purpose languages and dialects were being used, and the majority of these languages were used in embedded computer systems. This lack of programming language commonality contributes to the rising software costs in several ways (Ref 4:26):

- (1) it requires duplication in training and maintenance for the languages, their compilers, and their associated software support packages.
- (2) it limits communication among software practitioners.
- (3) it results in support software being developed which can only be used on one project.
- (4) it ties software maintenance to the original developer.
- (5) it limits the development of support and maintenance software.
- (6) it limits the applicability of new support software.
- (7) it creates a situation in which the adoption of an

existing language by a new project can be more risky and less cost-effective than the development of a new programming language specialized to the project.

1.1.2 The High Order Language Working Group

In order to resolve the problems presented by the large number of languages, the DoD began a common high order language programming effort. To coordinate this effort, a High Order Language Working Group (HOLWG) was formed. This group consisted of representatives from the Army, Navy, Air Force, Marine Corps, Defense Communications Agency and Defense Advanced Research Projects Agency. The HOLWG was chartered to "investigate the establishment of a minimal number of common high-order computer programming languages to be used in the development, acquisition, and support of computer resources embedded within Defense Systems" (Ref 4:27).

1.1.3 The search for a solution

The first step taken by the HOLWG was to adopt an interim list of seven programming languages approved for use in the development of new defense system software. The second step was to determine the characteristics of a general-purpose programming language suitable for embedded computer applications. These characteristics were put in the form of requirements which were circulated to the military, industrial and academic communities for comments.

After the comments were received and evaluated, a new requirements list was circulated. This process was repeated until a suitable language description was defined.

The next step was an evaluation to determine if any existing language met the requirements specified in the language description. The results obtained from evaluations of 23 different languages led the HOLWG to conclude that no existing language satisfied the requirements well enough to be adopted as a common language. Even though no existing language was found suitable, the evaluators did agree that it was possible to design a single language that would meet all the requirements. Based on this finding, the HOLWG began directing their efforts toward the design of a new language (Ref 4:27).

1.1.4 Design Phase

The design phase evaluated fifteen design proposals received for the new language description. Of these, four were selected for parallel development efforts. At the conclusion of these efforts, the language definition developed by CII Honeywell-Bull was accepted as the basis for the new DoD language, now known as Ada (Ref 4:29).

1.1.5 The need for a standard

The key to the economic success of Ada is the portability of programs, programmers, compilers and software tools. To insure portability it was essential that Ada be

established as a clear and unambiguous standard. In addition a means for discouraging and detecting compilers which did not correctly implement the standard was needed. This led to the trademarking of the name Ada and the development of a means for validating compilers.

1.1.6 The Ada Compiler Validation Capability

The Ada Compiler Validation Capability (ACVC) effort began at SofTech in September of 1979, and the first version was completed in 1981. It is especially significant because it makes Ada the first programming language to have a means of enforcing the language specification before diverse implementations begin to appear (Ref 6:57).

The primary purpose of the ACVC was to determine if Ada compilers comply with the language definition contained in the "Reference Manual for the Ada Programming Language". The ACVC was also designed to help the implementers comply with the language standard, by pointing out potential implementation difficulties (Ref 6:57).

The current version of the ACVC has three main components (Ref 2:1-1):

1. An Implementers Guide (IG) which describes the implementation implications of the Ada standard and conditions which are to be checked by the validation tests.

2. Test programs that are submitted to the compiler being tested. The current version of the ACVC has over 1400 tests designed to check the compilers conformance to the language specification.

3. Validation support tools that are used to prepare tests for execution and to analyze the results of execution.

1.1.7 Ada compiler development

Widespread acceptance of the Ada programming language is not likely to occur until compilers become readily available. Currently there are a large number of compiler development efforts under way. The first successful validation of an Ada compiler was expected to occur in late 1982.

A large number of the compiler development efforts are being directed at the microcomputer market. The approach many of these efforts have taken is to develop a basic subset of the full Ada language. Once the subset is developed, enhancements are then added to it until the entire language is implemented (Appendix A provides a more detailed discussion of a typical development effort).

Developers of these subsets have encountered a large number of problems. Two of the major problems are the complexity of the language and the lack of adequate support tools needed during the compiler development process.

1.1.8 The need for an incremental validation capability

A significant aspect of the ACVC is that it is targeted to test only those compilers which are completed and implement the entire language. As a result, the ACVC uses language features which are likely to be supported only toward the end of the compiler construction. This severely limits the usefulness of the ACVC during the development phases of a compiler. This is significant because the cost of repairing an error is reduced if it is detected soon after it is committed. This makes the availability of adequate test tools for compilers essential during the developmental stages. Errors made during the early stages of the compiler development could be extremely costly if they go undetected until attempts are made to validate the compiler.

1.2 OBJECTIVE

The objective of this project was to develop techniques for transforming the existing ACVC test set into a version which could be used to test evolving Ada compilers.

1.3 GENERAL APPROACH

The approach taken in this project was significantly influenced by the requirement for Ada compilers to pass the ACVC. As a result of this requirement, the project was directed toward modifying the current set of ACVC tests rather than attempting to develop a completely new test set.

The approach taken was to remove any language-features used in the test set that are not supported by the compiler being tested. Once the unsupported features are removed, the compiler should pass the remaining portion of the test set. Automating the process which removes unsupported features from the tests would allow a new test set to be developed every time new features are added to the compiler.

The advantage of this approach is that the actual ACVC tests are being used. Tests are incorporated into the test set as soon as the language-features used by the test are supported by the compiler. The test set will continue to grow as the compiler becomes more complete. Passing tests or portions of tests used in the ACVC should provide some degree of confidence that the actual ACVC tests can be passed. It will also help point out deficiencies in the compiler early in the development stages.

1.4 SEQUENCE OF PRESENTATION

The project consisted of three major phases, which are described in the following chapters. The first phase was an analysis of the existing ACVC. This phase studied the different types of tests contained in the ACVC and identified the use of language features in the tests which were not related to the test objectives (language features that are used unnecessarily will be referred to as language-feature dependencies). The analysis phase then looked for ways the tests could be repackaged without the

language-feature dependencies.

The second phase of the project investigated the problem of transforming the test set into a version that could be used to test evolving compilers. This focused on the removal of features not yet supported in an evolving compiler. Particular emphasis was placed on the automation of a process to accomplish this.

The third phase was a partial implementation of the tool described in the second phase. The purpose of the partial implementation was to demonstrate that it was possible to automate the removal of unsupported language features.

2. ANALYSIS

2.1 Introduction

The first step in this project was a detailed analysis of the existing ACVC test set. The purpose of this analysis was to identify the language-feature dependencies contained in the ACVC test set and determine what impact their removal would have on the test set. This requires a basic understanding of the ACVC's approach to testing, its design goals, its organization, and the general structure of the test set.

This analysis is broken into four sections. The first section provides background information on the testing approach, the design goals, and the organization of the test set. The second section looks at the the general structure of the test set, while the third section provides a detailed look at some of the tests found in the test set. The fourth section summarizes the results of the analysis.

2.2 Background

The first step in the analysis of the ACVC was to review the test set. This review looked at the testing approach taken by SofTech, some of the factors that influenced the design of the test set, and the organization of the test set.

2.2.1 Approach to testing

Black-box and white-box testing are the two generally accepted approaches to testing. White-box testing is predicated on a detailed knowledge of the internal workings of a product. Tests are designed to determine if internal operations are performed according to specification. Black-box tests, on the other hand, are designed to demonstrate that the functions a product is supposed to perform are operational. The internal structure of the software is not considered when designing black-box tests (Ref 7-292).

The designers of the ACVC used the black-box approach to testing. This was appropriate since the ACVC was designed to determine only if the compiler conformed to the language definition. Issues such as quality and efficiency were not considered when designing the test set. Also, to use the white-box approach would have required a detailed knowledge of each compiler submitted for validation. Since the manner in which various tasks are implemented may differ greatly between compilers, a new test set would be required for each compiler submitted for validation.

2.2.2 Test set design

Before making any changes to the test set several factors which influenced its design must be considered. This section will briefly describe some of the design goals of the test set and some of the factors which had a significant

influence on the design of the test set.

One of the ACVC's principle design goals was to develop a test set which was portable. To accomplish this, SofTech adopted the following set of coding standards in their test set (Ref 3:A-2):

- (1) The source line length in test programs does not exceed 72 characters.
- (2) Tests are limited to the basic 55 character set.
- (3) Numeric values were limited so that a 12 bit word size is sufficient.
- (4) Array sizes are kept small.
- (5) No tests use both fixed and floating-point types unless the test objective addresses interactions between these types.
- (6) Unnecessary use of fixed and floating-point types, integer types other than INTEGER, access types, tasks, generics, representation specifications, subunits, exceptions, overloading, renaming, private types, and input/output is prohibited.

In addition to insuring that the test set is portable, these coding standards also help reduce the number of failures that are not related to the test objectives (Ref 6:60).

Another design goal was to reduce the manual intervention required when using the test set. It resulted in the development of a large number of small tests which

require no modification during the testing process (Ref 6:59).

The need for the ACVC to be constantly updated also impacted on its design. Black-box tests cannot guarantee that software is error free. Therefore as errors are found in compilers which successfully passed the validation test, new tests must be developed to insure that these errors are identified in future validation attempts. This was another reason the use of small tests was adopted in the test set (Ref 6:59).

Finally, the decision to test only completed compilers was significant since it allowed features normally supported only towards the end of a compiler's development to be used. The prime example of this is a separately compiled package used to report test results (Report Package).

2.2.3 Test Set Organization

The tests in the test set are organized to correspond with objectives contained in the Implementers' Guide. These objectives can be broken into eleven major areas:

1. Lexical Elements
2. Declarations and Types
3. Names and Expressions
4. Statements
5. Subprograms
6. Packages
7. Visibility Rules
8. Tasks
9. Program structure and compilation issues
10. Exceptions
11. Generic program units

The Implementers' Guide was designed to correspond with the "Reference Manual for the Ada Programming Language" (LRM). Objectives found in the IG were based on the language definition contained in the LRM.

The language definition was reviewed to identify potential implementation difficulties. A set of test objectives was then developed to insure these potential deficiencies were identified during testing. The final step was to develop tests which would accomplish the test objectives.

Figures 2.1 and 2.2 reflect the relationship that exists between the LRM and the IG. Figure 2.1 is an example of the language definition taken from the LRM, while Figure 2.2 is a list of the test objectives taken from the IG that correspond to the language definition in Figure 2.1.

The language definition presented in the LRM uses a simple variant of the Backus-Naur form (BNF). The BNF is used to specify the rules for forming valid programs. These rules are called productions. Figure 2-1 shows the productions that define an identifier in Ada. The LRM uses square brackets to enclose optional items, braces to identify items which may occur repeatedly (zero or more times), and vertical bars (|) to separate alternative items.

2.3 Identifiers

Identifiers are used as names (also as reserved words). Isolated underscore characters may be included. All characters, including underscores, are significant.

identifier ::=

letter { [underscore] letter_or_digit }

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter

Note that identifiers differing only in the use of corresponding upper and lower case letters are considered as the same.

Figure 2-1. LRM definition of identifier structure (Sec 2.3)

The productions shown in figure 2-1 show that identifiers must begin with a letter. They also show that consecutive underscores are not permitted in identifiers, and that identifiers cannot end with an underscore.

Test Objectives and Design Guidelines

1. Check that upper and lower case letters are equivalent in identifiers (including reserved words).

Implementation Guideline: Try some all-upper, all-lower, and mixed case identifiers.

2. Check that consecutive, leading, and/or trailing underscores are not permitted in identifiers.
3. Check that identifiers can be as long as the maximum input line length permitted by the implementation and that all characters are significant (e.g., not just the first 8 or 16, or not just the first m and last n characters). Try identifiers serving as variables, enumeration literals, subprogram names, parameter names, entry names, record component names, type names, package names (both library units and subunits), statement labels, block labels, loop labels, task names, and exception names.

Implementation Guideline : Maximum length subprogram names and package names should be checked in separate tests.

4. Check that ? % @ # ' are not permitted in identifiers.

Figure 2-2. ACVC test objectives and design guidelines (section 2.3)

2.3 General Structure

The second step in the analysis looked at the structure of the test set. The test set has two primary components, the Report Package and the tests. This section will present an overview of each of these components.

2.3.1 Report Package

The report package is a separately compiled group of routines used to automate the process of reporting test results. They are independent of the tests themselves, and provide the mechanism for reporting pass/fail results of executable tests (Ref 3:A-1).

The report package (see Appendix B for the source listing) contains the following subprograms (Ref 3:B-1):

1. Test: This procedure is called at the beginning of all executable tests. It saves the test name and outputs the name and description.
2. Failed: This procedure outputs a failure message that includes a brief description of what failed.
3. Comment: This procedure outputs a comment message.
4. Result: This procedure is called at the end of each test. It indicates whether whether the test has passed or failed.

5. Put_Msg: This procedure formats and outputs messages. It can only be called within the package itself.

6. Ident_int, Ident_char, Ident_bool, and Ident_str: These functions are dynamic value routines which serve as identity functions for the types Integer, Character, Boolean and String.

7. Equal: A recursive equality function for the type integer.

2.3.2 Tests

The tests in the ACVC are written to correspond to the objectives listed in the Implementers' Guide. There are six distinct classes of tests which may be found in the test set (Ref 6:60):

Class A: These tests are designed to compile and execute without any errors. No checks are made at run-time to determine if a test objective has been met.

Class B: These tests are illegal and should fail compilation. They are passed if all errors are detected at compile time and all legal statements are considered legal.

Class C: These tests are designed to compile and execute. They are self-checking.

Class D: These are capacity tests. There are no pass/fail criteria.

Class E: These are tests that check whether certain implementation dependent options have been provided. They also determine how ambiguities in the language standard have been resolved.

Class L: These are illegal programs that are expected to fail at link-time. The failure must occur before any declarations in the main program or any units referenced in the main program are elaborated. They may fail compilation in some implementations.

2.4 Detailed Analysis

This section will provide a detailed analysis of each type of test found in the test set. It will specifically address four questions:

- (1) What language-feature dependencies exist ?
- (2) What effect will their removal have ?
- (3) How can the test be repackaged to eliminate the language-feature dependencies ?
- (4) Can the repackaged tests be used to test evolving compilers, or are further modifications needed ?

2.4.1 Class A

As mentioned before, Class A tests are designed to compile without any errors. Approximately two percent of the tests found in the test set are Class A tests. A typical example of a Class A test is shown in figure 2-3.

The test shown in figure 2-3 uses several language features which do not contribute to the accomplishment of the test objectives. The first dependency is the WITH REPORT statement. It is used to indicate the dependency of the procedure on the Report Package. The second dependency is the USE REPORT statement, which acts like a declaration in the procedure. The final two dependencies are the procedure calls TEST and RESULT. Both of these procedures are found in the separately compiled REPORT package.

After identifying the dependencies, the next step is to determine what effect their removal would have on the test set. The first consideration is whether the test objectives will still be accomplished. Since the purpose of the statements containing the dependencies is to report the pass/fail status of the tests, there is no impact on the test objectives. Another consideration is whether the initial design goals are still accomplished. In this case the removal of the dependencies would result in a significant increase in the manual effort required to analyze test results. Therefore the initial design goals would not be accomplished.

```

-- A21001A.ADA
-- CHECK THAT THE BASIC CHARACTER SET IS ACCEPTED
-- OUTSIDE OF STRING LITERALS AND COMMENTS.
-- DCB 1/22/80

WITH REPORT;
PROCEDURE A21001A IS
    USE REPORT;

BEGIN
    TEST ("A21001A", "CHECK THAT BASIC CHARACTER SET IS ACCEPTED" );

    DECLARE

        TYPE TABLE IS ARRAY (1..10) OF INTEGER;
        A : TABLE := ( 2 | 4 | 10 => 1 , 1 | 3 | 5..9 => 0 ) ;
                                -- USE OF : ( ) | ,

        TYPE BUFFER IS
            RECORD
                LENGTH : INTEGER;
                POS : INTEGER;
                IMAGE : INTEGER;
            END RECORD;                                -- USED TO TEST . LATER

        R1 : BUFFER;
        ABCDEFGHIJKLM : INTEGER;    -- USE OF ABCDEFGHIJKLM
        NOPQRSTUVWXYZ : INTEGER;    -- USE OF NOPQRSTUVWXYZ
        Z_1234567890 : INTEGER;    -- USE OF _1234567890

        I1, I2, I3 : INTEGER;
        C1, C2 : STRING (1..6);
        C3 : STRING (1..12);

    BEGIN
        I1 := 2 * ( 3 - 1 + 2 ) / 2 ; I2 := 8 ; -- USES ( ) * + - / ;
        C1 := "ABCDEF" ; -- USE OF "
        C2 := C1;
        C3 := C1 & C2 ; -- USE OF &
        I2 := 16#D#; -- USE OF #
        I3 := A'LAST; -- USE OF '
        R1.POS := 3; -- USE OF .
        IF I1 > 2 AND
            I1 = 4 AND
            I1 < 8 THEN -- USE OF > = <
            NULL;
        END IF;
    END;
    RESULT;
END A21001A;

```

Figure 2-3. Class A Test

The next step is to look for a way the tests can be constructed without the language-feature dependencies. The problem is that there is a need for the capability to report test status, therefore there will always be language-feature dependencies. Since the dependencies cannot be completely eliminated, emphasis should be placed on developing a method for reporting the status using language features likely to be supported early in a compiler's development. Special emphasis must be placed on eliminating the need for separate compilation, since it is usually not implemented in the early stages of a compiler development effort.

```
procedure sample_test is
  procedure test is
    begin
      -- code
    end;
  procedure result;
    begin
      -- code
    end;
begin
  test;
  -- code
  result;
end;
```

Figure 2-4. Repackaged Structure for Class A tests

There are several ways the test set could be repackaged to reduce the language dependencies. Perhaps the simplest method is to insert the TEST and RESULT procedures into the actual test in the manner shown in figure 2-4.

This method would require the insertion of two procedures in the test and the elimination of the WITH and USE statements. Also the manner in which input/output is handled must be considered since the TEST and RESULT procedures require some means of input/output. In the REPORT package the dependency on TEXT_IO is declared. The manner in which input/output is handled in evolving compilers varies greatly, so the actual code for the TEST and FAIL procedures will probably need to be rewritten for each compiler being tested.

The final question considered is whether Class A tests could be used to test evolving compilers once the language-feature dependencies are removed. The answer depends on what language features are implemented by the compiler being tested, and what features are used in the tests. For instance, the test shown in figure 2-3 would fail if the compiler being tested did not support the record structure. This means for Class A tests to be usable, some means for identifying language features must exist. Tests that use features not supported by the compiler must either be eliminated from the test set, or the features not supported must be eliminated from the test. In the case of the test in figure 2-3, the type declaration of buffer, the declaration of R1 as type buffer, and the assignment statement `R1.pos := 3` could be removed. The resulting test could then be used to test evolving Ada compilers which do not support records.

```

-- C27001A.ADA

-- CHECK THAT A COMMENT IS TERMINATED BY THE END OF THE
-- LINE, AND NOT BY THE NEXT -- (ELSE THIS COMMENT WILL BE
-- TREATED AS CODE).

-- DCB 1/16/80

WITH REPORT;
PROCEDURE C27001A IS
    USE REPORT;

    I1 : INTEGER;
BEGIN
    TEST("C27001A", "COMMENTS TERMINATED BY END OF LINE");

    I1 := 5;    -- I1 INITIALIZED.

-- BELOW CHECKS THAT A COMMENT IS TERMINATED BY END OF LINE
    I1 := I1 + 7;
    IF I1 /= 12 THEN
        FAILED("COMMENTS NOT TERMINATED BY END OF LINE");
    END IF;

    RESULT;
END C27001A;

```

Figure 2-5. Class C Test

2.4.2 Class C

Class C tests are very similar to the Class A tests. The primary difference is that Class C tests have a run-time check, usually in the form of an IF statement. Approximately 65 percent of the tests in the test set are Class C tests. An example of a Class C test is shown in Figure 2-5.

Class C tests contain the same dependencies found in Class A tests. They also have one additional dependency, the procedure call FAILED. The FAILED procedure is also found in the separately compiled REPORT package.

The test objectives of Class C tests will not be accomplished if the FAILED procedure is removed. The FAILED procedure is an integral part of the run-time check, and if it is removed the run-time check will not be performed. Therefore it is essential that some method be developed for repackaging the FAILED package.

The method used to repackage Class A tests can also be used for Class C tests. The only major difference would be the inclusion of the FAILED procedure. The repackaged tests would have the form shown in figure 2-6.

```
procedure sample_test is
  procedure failed is
    begin
      --code
    end;
  procedure test is
    begin
      -- code
    end;
  procedure result;
    begin
      -- code
    end;
begin
  test;
  -- code
  result;
end;
```

Figure 2-6. Restructured Class C test

Class C tests are similar to Class A tests in that they cannot be used to test an evolving compiler unless all of the unsupported features are removed from the tests.

```

-- D4A002B.ADA

-- LARGER LITERALS IN NUMBER DECLARATIONS, BUT WITH RESULTING
-- SMALLER VALUE OBTAINED BY SUBTRACTION. THIS TEST LIMITS
-- VALUES TO 64 BINARY PLACES.

WITH REPORT;
PROCEDURE D4A002B IS
USE REPORT;

X : CONSTANT := 4123456789012345678 - 4123456789012345679;
Y : CONSTANT := 4 * (10 ** 18) - 3999999999999999999;
Z : CONSTANT := (1024 ** 6) - (2 ** 60);
D : CONSTANT := 9_223_372_036_854_775_807/20_303_320_287_433;
E : CONSTANT := 36_028_790_976_242_271 REM 17_600_175_361;
F : CONSTANT := ( -2 ** 51) MOD ( - 131_071 );

BEGIN TEST("D4A002B", "LARGE INTEGER RANGE (WITH CANCELLATION) IN " &
"NUMBER DECLARATIONS; LONGEST INTEGER IS 64 BITS ");

    IF X /= -1 OR Y /= 1 OR Z /= 0
    OR D /= 454_279 OR E /= 1 OR F /= -1
    THEN FAILED("EXPRESSIONS WITH A LARGE INTEGER RANGE (WITH " &
"CANCELLATION) ARE NOT EXACT ");
    END IF;
    RESULT;
END D4A002B;

```

Figure 2-7. Class D Test

2.4.3 Class D

Class D capacity tests make up less than one percent of the tests in the test set. They have the same structure as the Class C tests, and they contain the same dependencies. Figure 2-7 shows a typical Class D test. Class D tests should be repackaged in the same manner as the Class C tests.

```

-- B32A06A.ADA

-- CHECK THAT IDENTIFIERS IN SEPARATE OBJECT_DECLARATIONS
-- WITH IDENTICAL ARRAY_TYPE_DEFINITIONS ARE DIFFERENT
-- TYPES.

-- DAT 3/17/81

PROCEDURE B32A06A IS

    A : ARRAY (BOOLEAN) OF BOOLEAN;
    B : ARRAY (BOOLEAN) OF BOOLEAN;

BEGIN

    A := B;                                -- ERROR: TYPE MISMATCH.
    A := (TRUE, TRUE);                     -- OK.
    IF A = B THEN                           -- ERROR: TYPE MISMATCH.
        NULL;
    END IF;

END B32A06A;

```

Figure 2-8. Class B Test

2.4.4 Class B

As mentioned before, Class B tests are designed to fail compilation. They make up approximately 25 percent of the tests. A typical Class B test is shown in Figure 2-5.

The review of Class B tests did not identify any language-feature dependencies. Class B tests do not use the REPORT package. Unlike Class A and Class C tests, the Class B tests do not require that unsupported features be removed from tests. Class B tests should still fail in any implementation.

```

-- LA3004A4.DEP

-- WKB 7/13/81

PACKAGE LA3004A4 IS
END LA3004A4;

WITH PKG;
PACKAGE BODY LA3004A4 IS

I : INTEGER := 4;

BEGIN

    PKG.P (I);

END LA3004A4;

```

Figure 2-9. Class L Test

2.4.5 Class L

The Class L tests were designed to fail at link time. Figure 2-8 shows a Class L test. The Class L tests make up approximately 7 percent of the tests in the test set.

These tests use separately compiled packages and many of the advanced features of the language. The tests also rely on other Class L tests to accomplish their objectives. Because of the relationships that exist between Class L tests, the changes made in one test may affect several other tests. It is unlikely that evolving compilers would find these tests useful. Because of the relationships that exist between these tests and their limited usefulness during compiler development, no attempt will be made to repackage these tests.

2.4.6 Class E

There are no examples of class E tests in the version of the ACVC being studied, therefore no analysis of a Class E test was made.

2.5 Summary

The analysis of the ACVC test set revealed that several language-feature dependencies existed in the test set. These dependencies could not be totally eliminated, but it was possible to repackage the tests to minimize the impact of the dependencies. The analysis also revealed that the elimination of dependencies is not enough to produce a test set which can be used to test subset compilers.

In addition to repackaging the tests, some method for removing unsupported language features is needed for Class A, Class C, and Class D tests. Once these unsupported features are removed, the Class A, Class C, and Class D tests could be combined with the Class B tests to make a viable test set.

3. PROJECT DEVELOPMENT

3.1 Introduction

This chapter describes the development of an automated tool designed to remove unsupported language-features from tests contained in the ACVC test set. The description begins with a brief overview of the development process. This is followed by three sections which describe the development of the tool's major components. The final section summarizes the development process.

3.2 Overview

The analysis presented in chapter 2 concluded that Class A, Class C and Class D tests must be modified before they can be used to test evolving Ada compilers. The analysis identified two modifications that must take place, the elimination of language-feature dependencies and the removal of language features not supported in the compiler being tested.

The removal of language-feature dependencies was discussed in chapter 2. It requires the recoding of three procedures contained in the separately compiled REPORT package. These recoded procedures must be inserted in the test programs.

The removal of unsupported features from the test set is a more difficult task. The fundamental system model of the process required to accomplish the task is illustrated in figure 3-1.

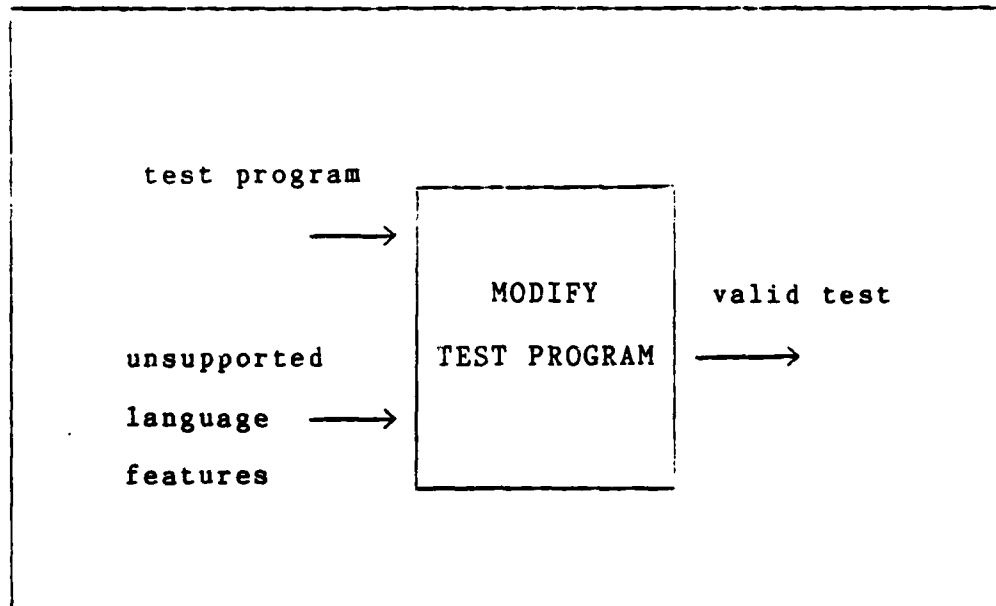


Figure 3.1 Overview of Test Development Process

The modification of the test program shown in figure 3-1 can be broken into three steps (Figure 3-2). The first step takes the test program as input and produces the productions and any identifiers, characters, numbers and strings as output. The second step uses the output from the first step to build a representation of the test program. The third step takes the representation and the list of unsupported features as input, and produces a modified program as output.

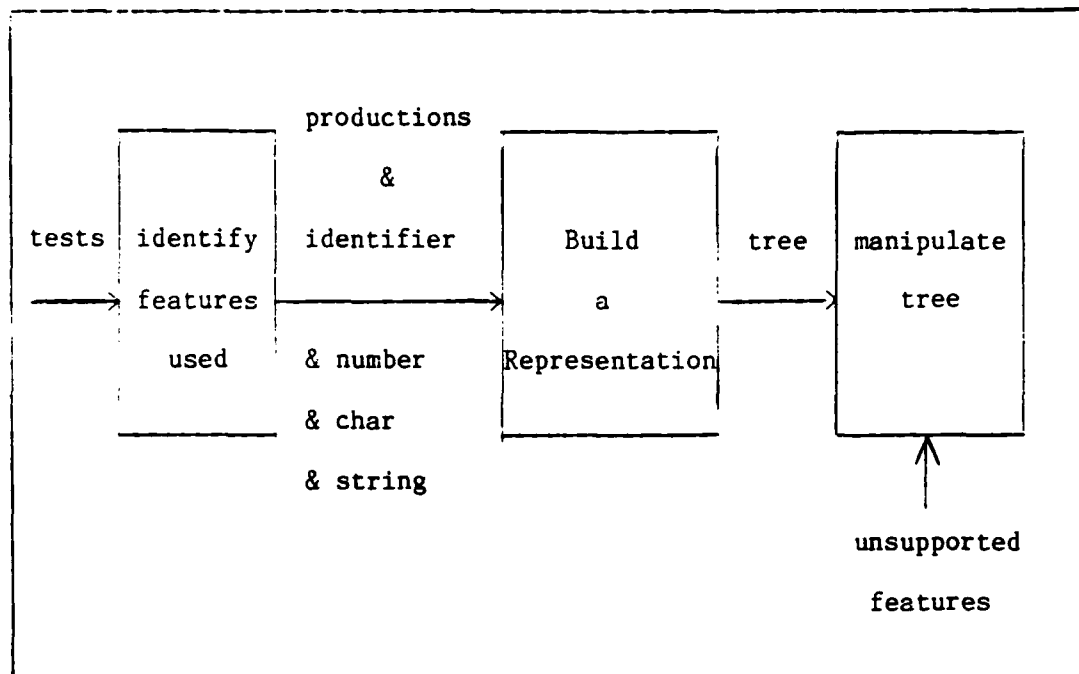


Figure 3-2. Internal structure of Modify Test Program

3.3 Identification of language-features

One of the problems encountered during the analysis phase was the need to identify the language-features used in the ACVC tests. Initially the identification was done manually using the BNF contained in the "Ada Reference Manual". In this approach, a matrix was produced that represented the tests contained in the test set and the language-features used in each of the tests. Once a compiler was identified for testing, the matrix would be used to identify tests which contained features not supported by the compiler. These tests would then be removed from the test set. This approach was abandoned for two principle reasons:

(1) The analysis indicated that tests containing unsupported language features could still be used if the unsupported features were removed (Appendix D contains an example).

(2) The amount of effort required to generate the matrix manually was unreasonable (Appendix E contains an example of a manual evaluation of a test).

The results of the first approach made it clear that any process developed to modify the test set would have to be completely automated to be of any value. Any process which required manual evaluation of the tests would take too much effort.

If the process for modifying tests was to be automated, the first step would still require the identification of the language-features used in the test set. Since the manual evaluation was essentially the same process performed by a parser, the possibility of using an existing parser was investigated. This investigation led to the selection of the parser used in a compiler developed by Alan R. Garlington (Ref 5) for use in the project (Appendix F describes Garlington's compiler).

The parser used in Garlington's compiler was selected for two primary reasons. The first reason was that it parsed the entire Ada language proposed in the 1980 version

of the LRM, which meant it should be capable of parsing all the tests in the ACVC test set. The second reason was that it was available in source code. This meant that it would be possible to modify the output from the parser based on the needs of the tool. An additional advantage was that the Garlington compiler was an evolving compiler and a potential candidate for testing. The Garlington compiler resided on the DEC-10 computer located at the Air Force Wright Avionics Laboratory. The decision was made to transport the Garlington compiler onto the VAX 11-780 at the Air Force Institute of Technology, where several other ongoing Ada efforts were residing. Appendix G describes the changes made to the Garlington compiler in order to get it to compile on the Vax 11-780.

Once the compiler was transported onto the VAX 11-780, efforts were then directed toward finding a method for outputting the language-features identified by the parser. The Garlington compiler has a switch (traceparse) which enables the output of productions used by the program, along with some other information. By modifying some write statements it was possible to output the valid productions. The only other output required from the compiler was the identifiers, characters, strings and numbers identified. Although it was determined this information could be extracted from the parser the effort did not go beyond identifying the form of the output.

The format for the output from the parser is as follows:

```
production 3
id = aname
string = "abcde"
number = 1234
char = 'a'
```

There are two reasons the efforts in the area of the parser were limited. First, changes made in the LRM meant the BNF used in Garlington's compiler (Appendix C contains the BNF used in Garlington's compiler) no longer complied with the Ada standard. Before a finished product could be developed a new parser would be needed. Second, the primary purpose of this thesis was to develop methods for repackaging ACVC tests so they could be used for testing evolving compilers. Therefore, the primary emphasis was placed on developing the process for modifying tests rather than on the modification of an existing parser. As the language continues to evolve the number of parsers available should increase.

3.4 Development of a representation

Before any modifications to a test program can be made some method of representing the program is required. A commonly used representation is a parse tree (Figure 3-3 shows the structure of a typical parse tree). The parse tree structure shown in figure 3-3 was considered, but it was not

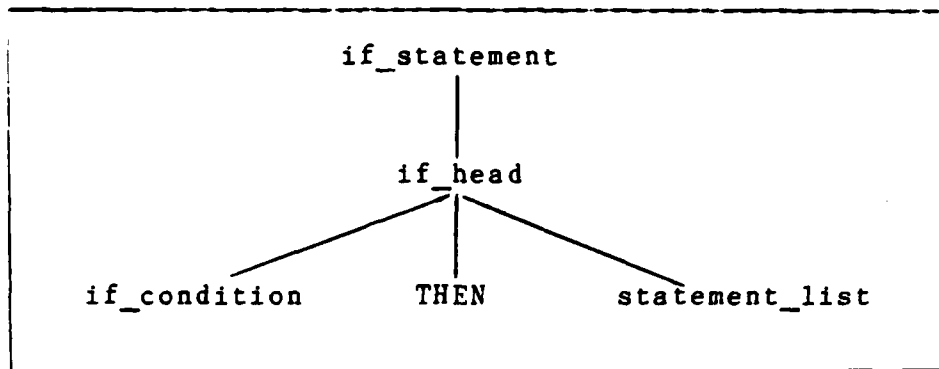


Figure 3-3. Parse Tree Representation

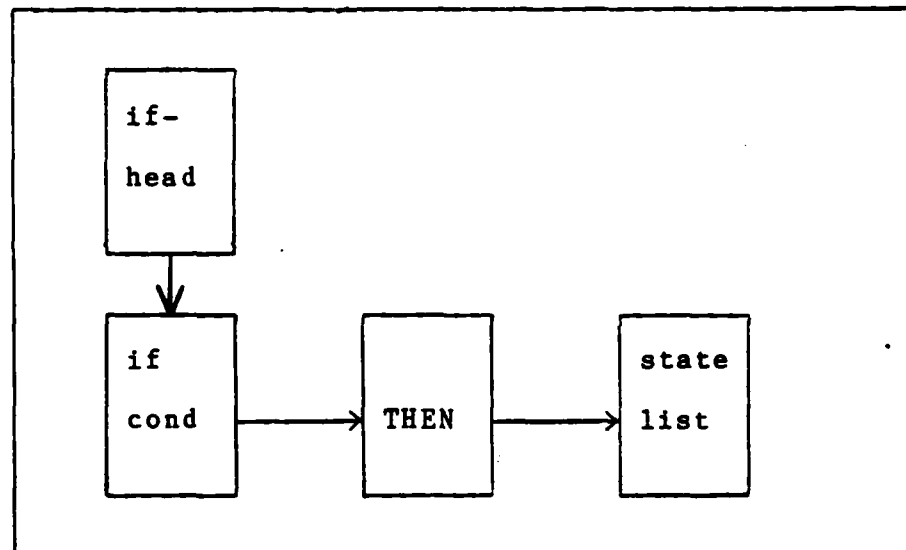


Figure 3-4. Representation of a production

used in this project. The structure used is shown in figure 3-4. It was selected because it would allow the use of well known binary tree traversal routines. It was also selected because it more accurately portrays the relationship between siblings, which becomes very important when it is necessary to remove language features from the tree.

The development of the representation can be broken into three steps. The first step reads the input file and creates separate lists to hold productions, identifiers, characters, strings and numbers. The input type is then identified and inserted at the head of the appropriate list.

The second step takes the production number from the head of the production list and builds a representation of the production. This representation must contain the production name and the production number. The parent node is given its actual production number, while its children are initialized to 0. Figure 3-5 shows what the structure would look like if the first production number was 3. The production number is shown in the upper left corner of each box. Nodes which contain printable tokens are identified with a 'P' in the lower right corner of the box.

The final step inserts the representation developed in the tree. This is accomplished using a post-order traversal beginning with the right (sibling) branch. The traversal searches the tree for a production name matching the one to be inserted. If a match is found the production number is checked. If the production number is 0 the representation is inserted, otherwise the search is continued. This check is necessary since the same production name may appear several places in the tree. This process is repeated until all productions have been inserted in the tree. Figure 3-6 shows what the tree would look like if the second production number was 4, and it was inserted in the tree.

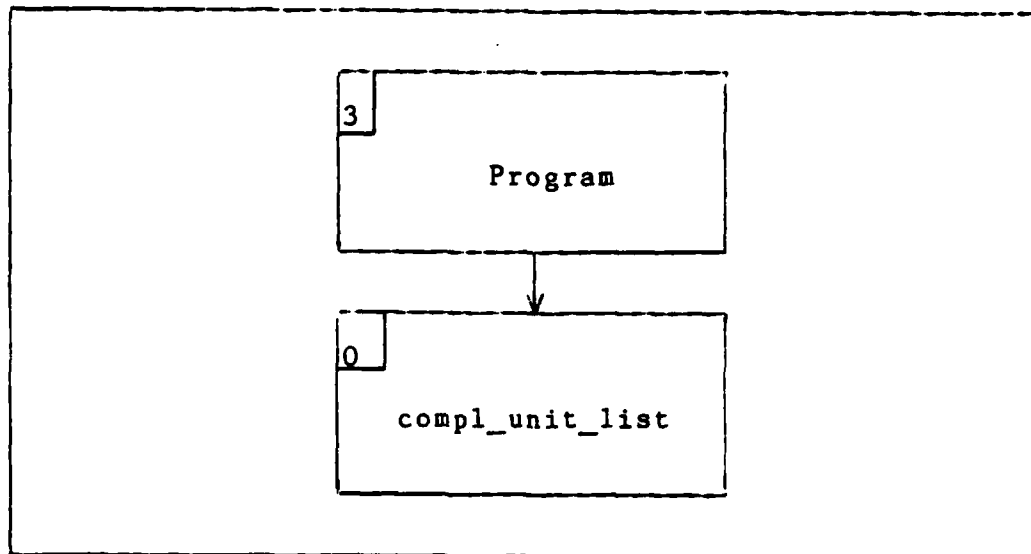


Figure 3-5. Representation of $\text{Program} ::= \text{compl_unit_list}$

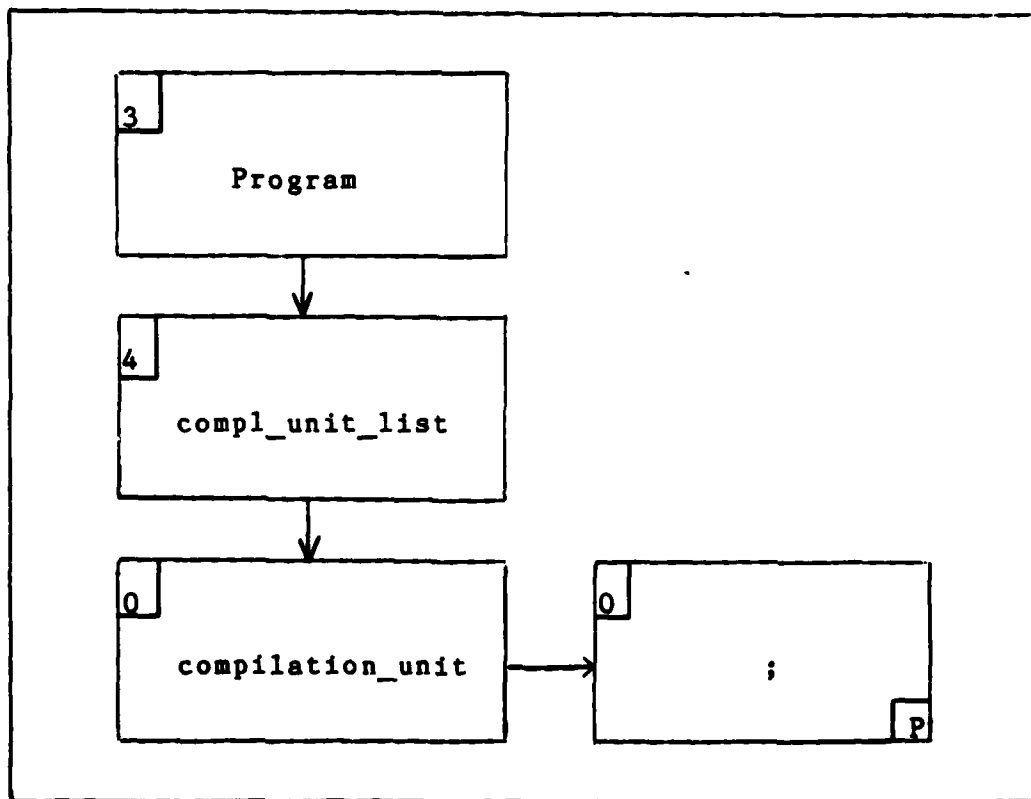


Figure 3-6. Representation after Production 4 is added

3.5 Removal of unsupported features

Once the representation of the test program is built, the next step is to remove the unsupported features from it. At first it appeared this could be accomplished by traversing the tree and cutting unsupported productions by setting pointers to NIL. A close look at sample representations showed this was not the case. There were two types of productions which required special treatment. The first type was the recursively defined productions (Appendix I), while the second type was the empty productions (Appendix H).

The most commonly encountered example of a recursive production is the `statement_list`.

```
206. statement_list ::= statement_list statement ;
```

Figure 3-7 shows an example of a tree representation which contains a `statement_list` production. If the compiler being tested does not support the NULL statement, the pointer to the NULL statement would be set to NIL. The tree left remaining is not a valid representation of an Ada program. For example, there could be a label left hanging on the tree. That means that the pointer labelled D must also be set to NIL. Another problem is that the comma following the statement is still left on the tree. Since the statement has been eliminated, the comma must also be removed. By setting the pointer labelled E to NIL, the comma would be

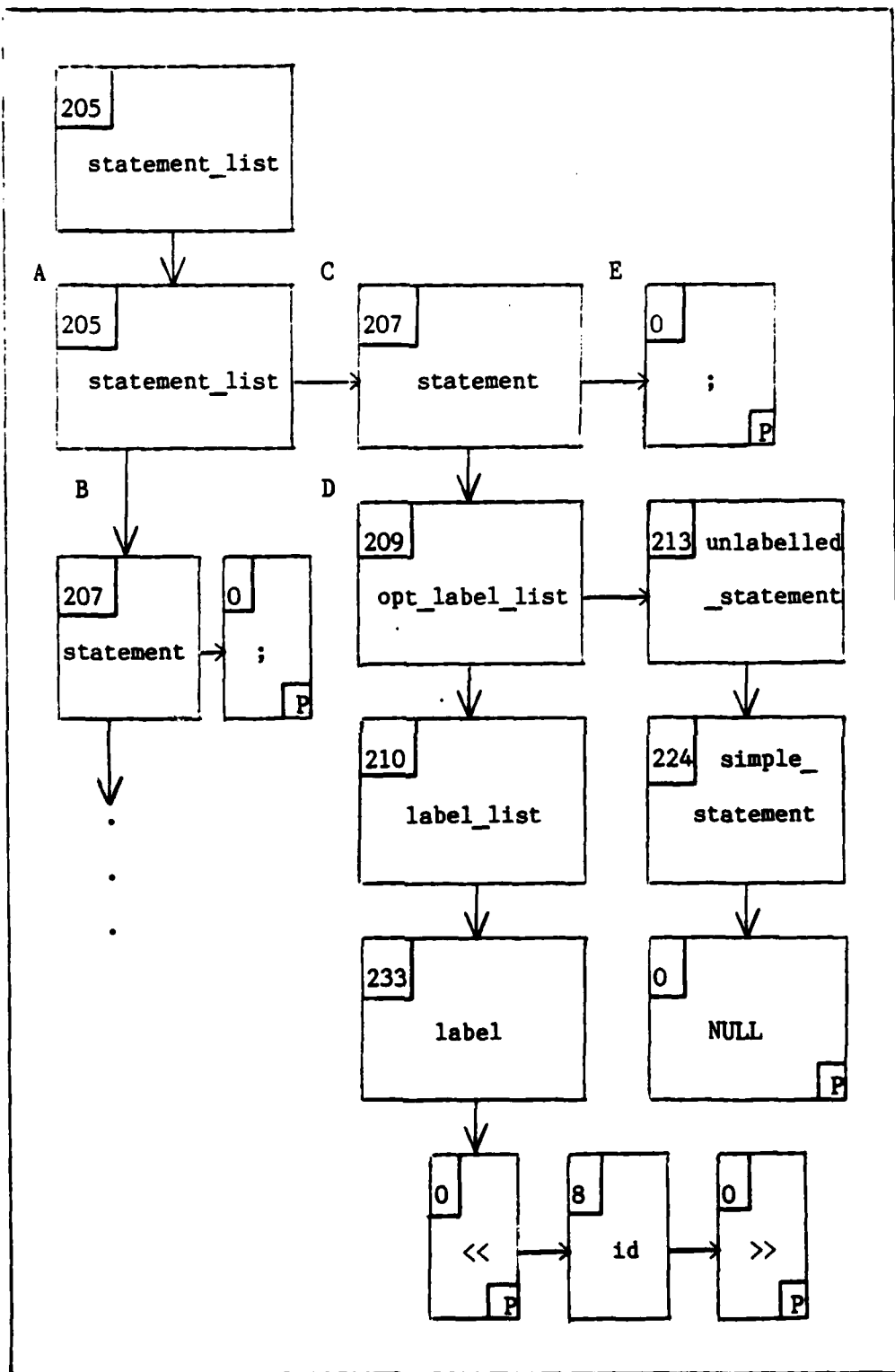


Figure 3-7. Representation of a recursive production

eliminated. While this completes the removal of the printable tokens which would be illegal in a program, it does not leave a valid tree. There are still nodes which should be removed. The statement node no longer belongs in the tree. To produce a valid representation requires setting the pointer labelled C (Figure 3-7) to NIL. It also requires that the pointer labelled A must be set equal to the pointer labelled B.

A slightly different case arises if the statement pointed to by the B pointer is not supported. The recursive traversal used would set the pointer C to NIL before the pointer to A is set equal to C. Therefore it is necessary to mark the descendant of any potentially recursive production. This stops the traversal before valid statements are eliminated.

The second type of productions encountered can be classified as empty productions (production 284 is an example). They are productions which contain no data in their children. Empty productions occur where the inclusion of information is optional in a program. The problem is identifying those instances where data is optional. Production 285 is an example of such a case. Designators may be included in programs, but there is no requirement for them. If they are removed from the program the remaining program is still valid.

```
284. designator_option ::=                --empty
```

```
285. designator_option ::= designator
```

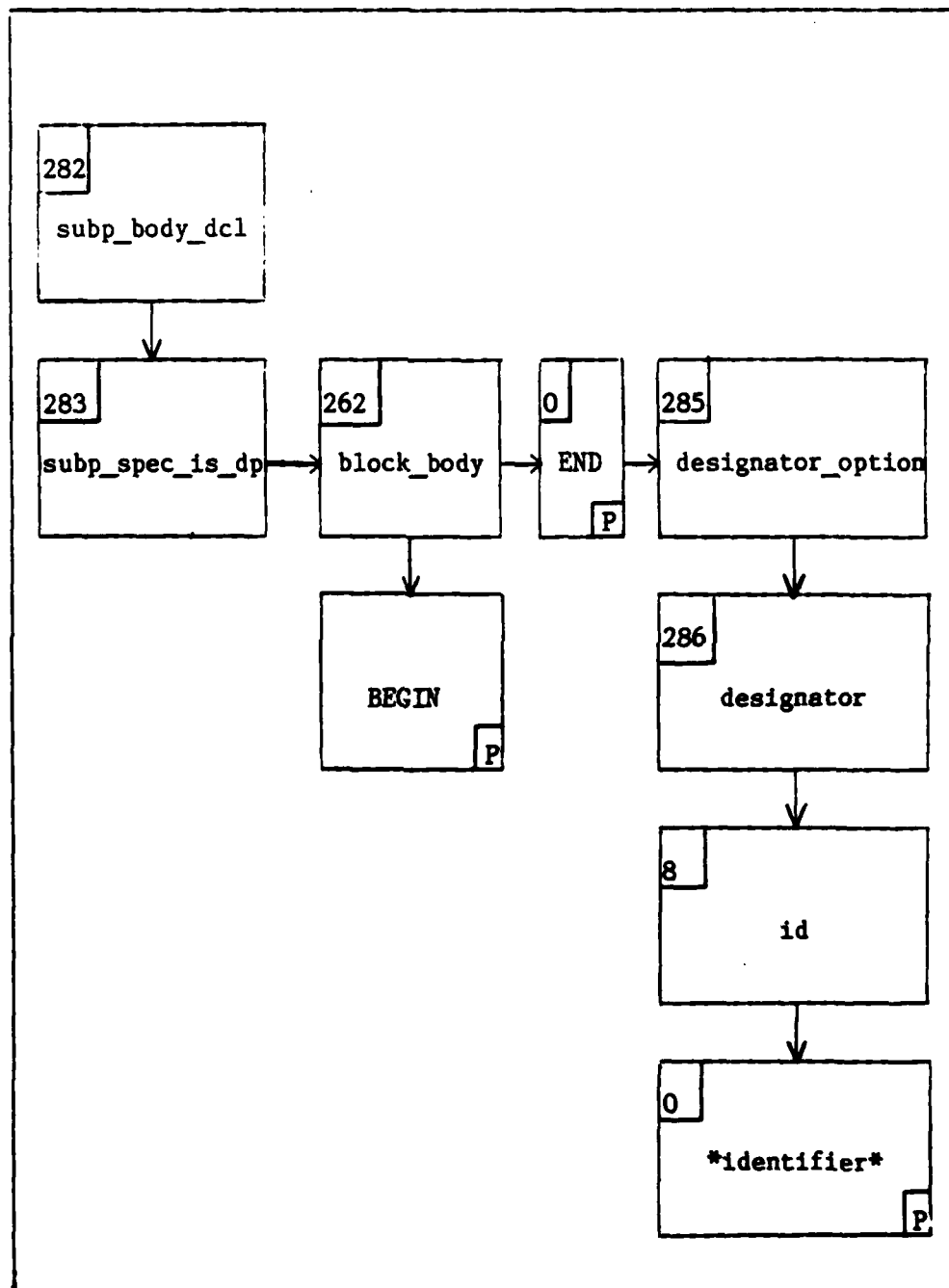


Figure 3-8. Representation of a potentially empty production

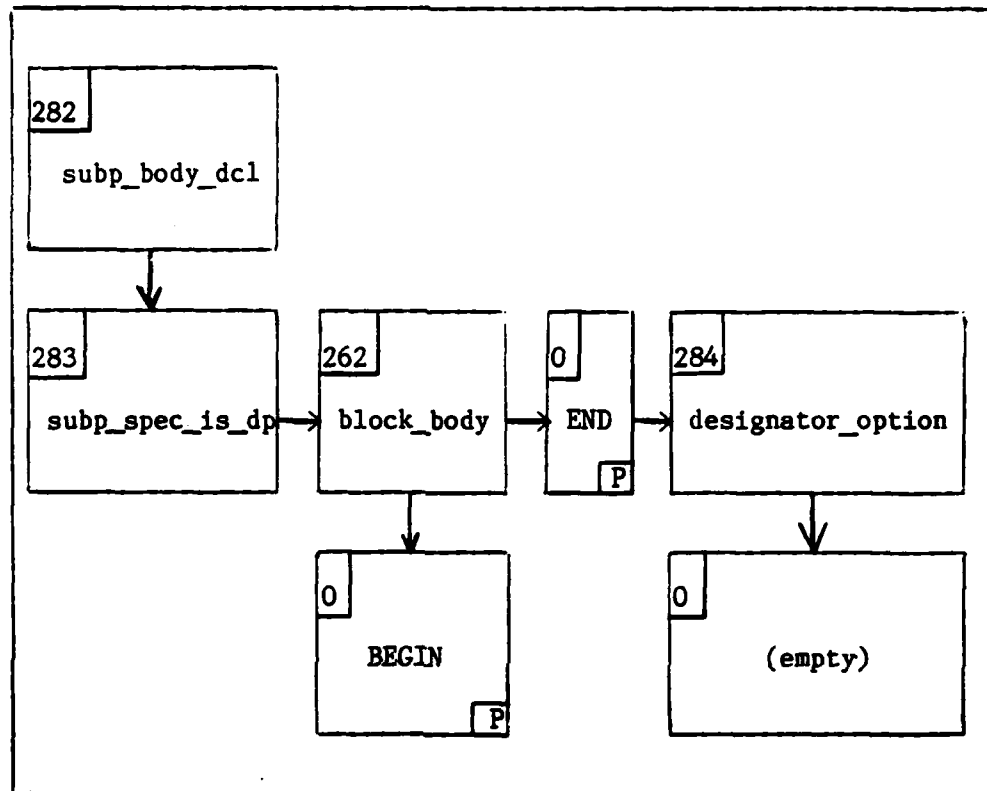


Figure 3-9. Representation after production is eliminated

Figure 3-8 shows an example of a structure using a production which is potentially empty. If production 285 was not supported by the compiler being tested, the only action required is to remove the data in the representation pointed to by its child pointer. In the representation shown in figure 3-8, the designator representation would appear as an empty box after the appropriate action was taken (figure 3-9 shows what the modified representation would look like). The

important fact to remember when eliminating the potentially empty productions is that the impact is localized. No changes are required at a higher level in the tree than where the empty node is.

3.5.1 Outputting the modified representation

After the unsupported features have been removed from the tree, the final step is to output the information contained in the remaining tree. This tree should represent a valid Ada program which can be submitted to the evolving compiler being tested.

Outputting the remaining representation is accomplished by traversing the tree in an in-order fashion beginning with the left (son) branch. Nodes in the tree which contain tokens that should be printed have the boolean variable PRINTABLE set to true. This was done in the record when the structure was being built.

3.6 Summary

The development of the project addressed several factors which must be considered when developing a tool to remove unsupported language-features. These included the specification of the input format, the method of representing the program, the different classes of productions which can be removed from the tree, and the method for outputting the modified tree. The next chapter will describe the tool actually developed and its capabilities.

4. IMPLEMENTATION

4.1 Introduction

Chapter three discussed the development of a tool which removes unsupported language-features from the test set. This chapter will discuss the actual implementation of the tool. It will describe the input required, the data structure used to represent the productions, the output from the tool, and the limitations on the use of the tool.

4.2 Input

The fundamental system model illustrated in figure 3-1 requires a list of unsupported language-features and a test program as input. The list of unsupported language-features must be created by the user and placed in a file named BADPRODS. In the current implementation the test program is submitted to the parser using the following command sequence:

```
px work.p < filename_of_test_program
```

The fundamental system model also shows productions, characters, strings, identifiers, and numbers as output from the parser and input to the tool. This input requires some manual preparation in the current implementation. This is a result of not having an adequate parser to work with. The format for the input was shown in chapter three. Productions are submitted in the order they are output from the parser. Identifiers, numbers, characters, and strings should be

placed in the same order they occurred in the program. Once this file is prepared it is submitted to the tool using the following command:

```
px tree.p < infile
```

There are two additional input files used by the tool, which should not require any modifications by the user. These files are called NULLPRODS and RECURSIVEPRODS. They contain a list of all productions which can be recursive or empty (these files can be found in Appendix H and Appendix I).

4.3 Data Structures

This section will describe the data structure used to represent the productions contained in the BNF. The data structure used is shown in figure 4-1. The fields in the record structure are used as follows:

DATA - contains the name of the production.

NUM - contains the production number (Appendix C).

SIBLING - points to a sibling.

SON - points to the scn (child).

PARENTPTR - points to the parent. This pointer is not currently being used. It can be connected by modifying the INSERTPRODUCTION procedure.

PRINTABLE - boolean used to identify the data in a node that should be output.

NEWLINE - boolean used for formatting purposes.

INDENT - boolean used for formatting purposes.

RECDISCENDANT - boolean used to identify a node as the son of a recursive production.

CUTNODE - boolean used to identify nodes which contain unsupported language features.

```
node = record
    data : dataarray;
    num : integer;
    sibling : nodeptr;
    son : nodeptr;
    parentptr : nodeptr;
    printable : boolean;
    newline : boolean;
    indent : boolean;
    recdescendant : boolean;
    cutnode : boolean;
end;
```

Figure 4-1. Data structure used to represent productions

The actual representation is generated by means of a large case statement. The case statement creates nodes containing the information shown in figure 4-1 for each production found in the BNF. The DATA field for characters, strings, numbers, and identifiers is filled by taking the top element from the corresponding list created by the parser. The RECDISCENDANT and CUTNODE fields are initialized false and then updated during the traversal of the tree by checking the BADPRODS and RECURSIVEPRODS lists.

4.4 Output

There are two options available for output. They are controlled by switches which are initialized in the INITGLOBALS procedure. The first switch is FULLTREE, which if set TRUE prints out the entire tree (minus comments). This option was provided primarily for testing purposes and would not normally be set TRUE. The second switch is CUTTREE, which if set TRUE will print out the modified version of the tree. All output is written to a file named OUTF.

Figures 4-2, 4-3 and 4-4 illustrate how the output from the tool should change as features are added to the compiler being tested. Fig 4-2 shows what a test should look like for a compiler which does not support arrays and records. Figure 4-3 shows what the new version of the test should look like after arrays are implemented. Figure 4-4 shows what the test should look like after records and arrays are both implemented.

4.5 Limitations

The current implementation has some limitations which the user should be aware of. Perhaps the most serious is the limitation imposed by the specific BNF used to develop the parser. It makes it difficult to identify and eliminate unsupported data types. For example, if integers were not supported, the BNF provides no way of determining whether a type is integer or real. The current BNF uses the following

productions to represent an integer:

subtype_indication ::= name

literal ::= number

There are two ways to overcome this problem. The first is by outputting more detailed information from the symbol table. The second and probably easier way is to use an extended form of the BNF which is more descriptive. The following extensions were made to the BNF to demonstrate types could be cut from the representation along with occurrences of the types in the program body.

subtype_indication ::= integer

literal ::= integer_number

integer_number ::= number

These extensions would be necessary for other data types as well. There is one problem not solved by the extensions, and that is the scope of the variables. This information would need to be obtained from the symbol table and stored in the record structure.

The process developed to modify the tests also imposes some inherent limitations. Modifications are based on syntactic issues and do not take into consideration the semantic rules being tested. No provisions are made in the test set to account for the semantic rules that an evolving compiler may not have implemented. An example of this would

```

WITH REPORT;
PROCEDURE A21001A IS
  USE REPORT;
BEGIN
  TEST ("A21001A", "CHECK THAT BASIC CHARACTER SET IS ACCEPTED" );
  DECLARE
    ABCDEFGHIJKLM : INTEGER;      — USE OF ABCDEFGHIJKLM
    NOPQRSTUVWXYZ : INTEGER;      — USE OF NOPQRSTUVWXYZ
    Z_1234567890   : INTEGER;      — USE OF _1234567890
    I1, I2, I3 : INTEGER;
    C1, C2 : STRING (1..6);
    C3 : STRING (1..12);
  BEGIN
    I1 := 2 * ( 3 - 1 + 2 ) / 2 ; I2 := 8 ; — USES ( ) * + - / ;
    C1 := "ABCDEF" ;                  — USE OF "
    C2 := C1;
    C3 := C1 & C2 ;                  — USE OF &
    I2 := 16#D#;                     — USE OF #
    IF I1 > 2 AND
       I1 = 4 AND
       I1 < 8 THEN                  — USE OF > = <
      NULL;
    END IF;
  END;
  RESULT;
END A21001A;

```

Figure 4-2. Test with records and arrays removed

be the length of identifiers allowed in the compiler being tested. The language definition allows identifiers to be as long as the maximum input line length permitted by the implementation. All characters in the identifier are significant. If the compiler developer chose to make only the first eight characters significant, no method is provided to modify the test set accordingly. It could be accomplished by generating shorter names for the identifiers found in the symbol table. This would make comparisons between the ACVC tests and the modified tests difficult.

```

WITH REPORT;
PROCEDURE A21001A IS
  USE REPORT;

BEGIN

  TEST ("A21001A", "CHECK THAT BASIC CHARACTER SET IS ACCEPTED" );

  DECLARE

    TYPE TABLE IS ARRAY (1..10) OF INTEGER;
    A : TABLE := ( 2 | 4 | 10 => 1 , 1 | 3 | 5..9 => 0 ) ;
                                — USE OF : ( ) | ,

    ABCDEFGHIJKLM : INTEGER;      — USE OF ABCDEFGHIJKLM
    NOPQRSTUVWXYZ : INTEGER;      — USE OF NOPQRSTUVWXYZ
    Z_1234567890 : INTEGER;       — USE OF _1234567890
    I1, I2, I3 : INTEGER;

    C1, C2 : STRING (1..6);
    C3 : STRING (1..12);

  BEGIN

    I1 := 2 * ( 3 - 1 + 2 ) / 2 ; I2 := 8 ; — USES ( ) * + - / ;
    C1 := "ABCDEF" ; — USE OF "
    C2 := C1;
    C3 := C1 & C2 ; — USE OF &
    I2 := 16#D#; — USE OF #
    I3 := A'LAST; — USE OF '
    IF I1 > 2 AND
      I1 = 4 AND
      I1 < 8 THEN — USE OF > = <
      NULL;
    END IF;

  END;
  RESULT;
END A21001A;

```

Figure 4-3. Test with record removed

```

WITH REPORT;
PROCEDURE A21001A IS
  USE REPORT;

BEGIN
  TEST ("A21001A", "CHECK THAT BASIC CHARACTER SET IS ACCEPTED" );

  DECLARE
    TYPE TABLE IS ARRAY (1..10) OF INTEGER;
    A : TABLE := ( 2 | 4 | 10 => 1 , 1 | 3 | 5..9 => 0 ) ;
                                — USE OF : ( ) | ,
    TYPE BUFFER IS
      RECORD
        LENGTH : INTEGER;
        POS : INTEGER;
        IMAGE : INTEGER;
      END RECORD;                                — USED TO TEST . LATER
    R1 : BUFFER;

    ABCDEFGHIJKLM : INTEGER;                    — USE OF ABCDEFGHIJKLM
    NOPQRSTUVWXYZ : INTEGER;                    — USE OF NOPQRSTUVWXYZ
    Z_1234567890 : INTEGER;                     — USE OF _1234567890
    I1, I2, I3 : INTEGER;

    C1, C2 : STRING (1..6);
    C3 : STRING (1..12);

  BEGIN
    I1 := 2 * ( 3 - 1 + 2 ) / 2 ; I2 := 8 ; — USES ( ) * + - / ;
    C1 := "ABCDEF" ;                        — USE OF "
    C2 := C1;
    C3 := C1 & C2 ;                          — USE OF &
    I2 := 16#D#;                             — USE OF #
    I3 := A'LAST;                            — USE OF '
    R1.POS := 3;                             — USE OF .
    IF I1 > 2 AND
      I1 = 4 AND
      I1 < 8 THEN                            — USE OF > = <
      NULL;
    END IF;
  END;
  RESULT;
END A21001A;

```

Figure 4-4. Complete test

5. RECOMMENDATIONS AND CONCLUSIONS

5.1 Introduction

This chapter describes areas where follow-on efforts could begin and where deficiencies in the current implementation exist. It also discusses conclusions reached about the feasibility of using the modified test set to test evolving Ada compilers.

5.2 Recommendations

There are several deficiencies in the current implementation which should be corrected before the tool is used to produce a valid test set. The most critical deficiency is the lack of an adequate parser. A parser needs to be developed using the current language description and any extensions that will allow a complete identification of the language-features used in the test set. This parser could possibly be built using LEX and YACC facilities provided on the VAX 11-780. The LEX and YACC facilities could conceivably be used to actually build the tree representation.

The development of a new parser using a different BNF will require a new case statement to generate the data structures. It is recommended that a program be written to automatically generate the case statement. This program should be able to generate the case statement using the BNF as input.

Once the parser is developed some method for keeping track of the scope of variables is needed. This problem may require storing symbol table information in the record structures used to represent the productions.

Another problem which needs to be addressed is the development of a method for automatically repackaging the tests. This will require either automating a text editing process or the use of some type of include command in the test skeleton developed.

Also, a driver needs to be written which will take the tests from the test set and feed them into the parser. It must then take the output from the parser and feed it to the tool implemented. The output from the tool must be concatenated with the other modified tests. The driver must be capable of identifying the test class by reading the test name. The user may consider doing this prior to submitting the tests.

Finally, the tool needs to be tested extensively. Because of the amount of time required to generate test cases by hand, the tool has not been thoroughly tested.

5.3 Conclusions

There were a couple of conclusions reached while working on this project concerning the quality of the ACVC and the value of a testing capability for evolving compilers. One conclusion was that the coding standards used by SofTech were very helpful when it became necessary

to analyze the tests. In most cases they used only the minimum amount of features needed to accomplish the test objectives. It appeared to be a well organized testing effort and should serve as an example for others to follow. Other compiler validation efforts studied were not nearly as well put together.

The final conclusion reached was that the subset testing capability will be of value to compiler developers when it is completed. Although it would not represent a complete testing capability, it is a good start to one. Also when completely implemented it would represent a very easy method for generating tests. The manual effort would be minimal.

BIBLIOGRAPHY

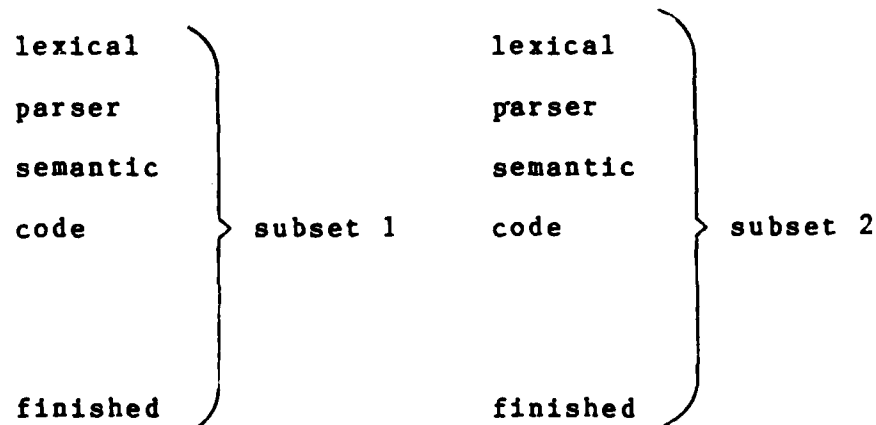
1. Defense Advanced Research Projects Agency. Reference Manual for the Ada Programming Language Proposed Standard Document. Washington, D.C. : Department of Defense, 1980.
2. Defense Advanced Research Projects Agency. Ada Compiler Validation Implementers' Guide. Prepared by SofTech, Inc. Waltham, Massachusetts. October 1980. (AD A091 760).
3. Defense Advanced Research Projects Agency. Ada Compiler Validation Test Programs. Prepared by SofTech, Inc. Waltham, Massachusetts. November 1981.
4. Fisher, David A. "DoD's Common Programming Language Effort," Computer, 11 (3): 24-33 (March 1978).
5. Garlington, Alan R. Preliminary Design and Implementation of an Ada Pseudo-Machine. MS Thesis. Wright-Patterson AFB, Ohio : Air Force Institute of Technology, March 1981. (AD A100 796).
6. Goodenough, John B. "The Ada Compiler Validation Capability," Computer, 14 (6): 57-64 (June 1981).
7. Pressman, Roger S. Software Engineering: A Practitioner's Approach. New York: McGraw-Hill Book Company, 1982.
8. Wetherell, Charles and Alfred Shannon. "LR Automatic Parser Generator and LR (1) Parser." Livermore, California: Lawrence Livermore Laboratory, 1979.

APPENDIX A

Evolving Compiler Development

This appendix describes the development of the evolving compilers the modified test set is targeted to test. It also provides a description of a typical Ada compiler development effort.

The compilers the modified test set is targeted to test are those which are developed as subsets. In other words, a subset of the language is defined and the compiler is built to compile just the subset. Additions are then made to the subset to produce a version closer to the complete language. The development process portrayed below is the type the test set is targeted for.



The compiler described below is typical of those targeted to be tested by the modified test set. It is the Ada/1000 Compiler developed by Science Applications, Inc. This information was taken from a Science Applications, Inc. advertisement.

The Ada/1000 compiler is scheduled to consist of four releases. The first release will support the following features:

1. Integer objects and operators
2. Boolean objects and operators
3. Nested procedures and functions
4. Simple user defined types (arrays, enumerations, simple records)
5. Sequential flow control (loop, if-then-else, case)

The second release of the Ada/1000 will provide these additional features :

1. Exception handling
2. Overloading
3. Packages (limited)
4. Separate compilation (limited)
5. Tasking (limited)
6. Attributes (limited)
7. Pragmas (limited)
8. Floating point objects and operators (limited)

The third release adds the following features:

1. Variant records
2. Full packages with separate compilation
3. Dynamic arrays (character strings and variants)
4. Representation specification (length and enumeration specification).
5. Tasking
6. Code statements
7. Derived types

The last release will be a validated Ada/1000 compiler. It will include the following features:

1. Generics
2. Fixed point objects and operators
3. Low-level I/O

As mentioned before, the development of the Ada/1000 compiler is typical of many of the current development efforts. The most significant aspect of these efforts is that separate compilation is not typically supported until the second or third release. This means the ACVC test set is of little use. It also means that the Input/Output package will most likely be non-standard.

APPENDIX B

Report Package

This appendix contains the report specification and the report body source listings found in the ACVC test set (Ref 3).

REPORT SPECIFICATION

-- REPSPEC.ADA
-- THE REPORT PACKAGE PROVIDES THE MECHANISM FOR REPORTING THE PASS/FAIL
-- RESULTS OF EXECUTABLE (CLASSES A, C, D, AND E) TESTS.
-- IT ALSO PROVIDES THE MECHANISM FOR GUARANTEEING THAT CERTAIN VALUES
-- BECOME DYNAMIC (NOT KNOWN AT COMPILE-TIME).
-- JRK 12/13/79
-- JRK 6/10/80
-- JRK 8/6/81

PACKAGE REPORT IS

-- THE REPORT ROUTINES.

PROCEDURE TEST

(NAME : STRING(1..7);
DESCR : STRING

);

-- THIS ROUTINE MUST BE INVOKED AT THE
-- START OF A TEST, BEFORE ANY OF THE
-- OTHER REPORT ROUTINES ARE INVOKED.
-- IT SAVES THE TEST NAME AND OUTPUTS THE
-- NAME AND DESCRIPTION.
-- TEST NAME, E.G., "C23001A".
-- BRIEF DESCRIPTION OF TEST, E.G.,
-- "UPPER/LOWER CASE EQUIVALENCE IN " &
-- "IDENTIFIERS".

PROCEDURE FAILED

(DESCR : STRING

);

-- OUTPUT A FAILURE MESSAGE. SHOULD BE
-- INVOKED SEPARATELY TO REPORT THE
-- FAILURE OF EACH SUBTEST WITHIN A TEST.
-- BRIEF DESCRIPTION OF WHAT FAILED.
-- SHOULD BE PHRASED AS:
-- "(FAILED BECAUSE) ...REASON...".

PROCEDURE COMMENT

(DESCR : STRING
);

-- OUTPUT A COMMENT MESSAGE.
-- THE MESSAGE.

PROCEDURE RESULT;

-- THIS ROUTINE MUST BE INVOKED AT THE
-- END OF A TEST. IT OUTPUTS A MESSAGE
-- INDICATING WHETHER THE TEST AS A
-- WHOLE HAS PASSED OR FAILED.

-- THE DYNAMIC VALUE ROUTINES.

-- EVEN WITH STATIC ARGUMENTS, THESE FUNCTIONS WILL HAVE DYNAMIC
-- RESULTS.

FUNCTION IDENT INT

(X : INTEGER
) RETURN INTEGER;

-- AN IDENTITY FUNCTION FOR TYPE INTEGER.
-- THE ARGUMENT.
-- X.

FUNCTION IDENT_CHAR	— AN IDENTITY FUNCTION FOR TYPE
(X : CHARACTER	— CHARACTER.
) RETURN CHARACTER;	— THE ARGUMENT.
	— X.
FUNCTION IDENT_BOOL	— AN IDENTITY FUNCTION FOR TYPE BOOLEAN.
(X : BOOLEAN	— THE ARGUMENT.
) RETURN BOOLEAN;	— X.
FUNCTION IDENT_STR	— AN IDENTITY FUNCTION FOR TYPE STRING.
(X : STRING	— THE ARGUMENT.
) RETURN STRING;	— X.
FUNCTION EQUAL	— A RECURSIVE EQUALITY FUNCTION FOR TYPE
(X, Y : INTEGER	— INTEGER.
) RETURN BOOLEAN;	— THE ARGUMENTS.
	— X = Y.
END REPORT;	

REPORT BODY

-- REPBODY.ADA

-- DCB 04/27/80

-- JRK 6/10/80

-- JRK 11/12/80

-- JRK 8/6/81

WITH TEXT_IO;

PACKAGE BODY REPORT IS

USE TEXT_IO;

TYPE STATUS IS (PASS, FAIL);

TEST STATUS : STATUS := FAIL;

TEST_NAME : STRING(1..7) := "NO_NAME";

PROCEDURE PUT_MSG (MSG : STRING) IS

-- WRITE MESSAGE. LONG MESSAGES ARE FOLDED (AND INDENTED).

MAX_LEN : CONSTANT INTEGER RANGE 50..150 := 72; -- MAXIMUM

OUTPUT LINE LENGTH.

INDENT : CONSTANT INTEGER RANGE 0..20 := 15; -- AMOUNT TO

INDENT CONTINUATION LINES.

I : INTEGER := 0;

CURRENT INDENTATION.

M : INTEGER := MSG'FIRST;

START OF MESSAGE SLICE.

N : INTEGER;

END OF MESSAGE SLICE.

BEGIN

LOOP

IF I + (MSG'LAST-M+1) > MAX_LEN THEN

N := M + (MAX_LEN-I) - 1;

IF MSG(N) /= ' ' THEN

WHILE N >= M AND THEN MSG(N+1) /= ' ' LOOP

N := N - 1;

END LOOP;

IF N < M THEN

N := M + (MAX_LEN-I) - 1;

END IF;

END IF;

ELSE N := MSG'LAST;

END IF;

SET_COL (I + 1);

PUT_LINE (MSG(M..N));

I := INDENT;

M := N + 1;

WHILE M <= MSG'LAST AND THEN MSG(M) = ' ' LOOP

M := M + 1;

END LOOP;

EXIT WHEN M > MSG'LAST;

END LOOP;

END PUT_MSG;

```

PROCEDURE TEST (NAME : STRING(1..7); DESCR : STRING) IS
BEGIN
    TEST_STATUS := PASS;
    TEST_NAME := NAME;
    PUT_MSG ("");
    PUT_MSG ("—— " & TEST_NAME & " " & DESCR & ".");
END TEST;

```

```

PROCEDURE COMMENT (DESCR : STRING) IS
BEGIN
    PUT_MSG (" - " & TEST_NAME & " " & DESCR & ".");
END COMMENT;

```

```

PROCEDURE FAILED (DESCR : STRING) IS
BEGIN
    TEST_STATUS := FAIL;
    PUT_MSG (" * " & TEST_NAME & " " & DESCR & ".");
END FAILED;

```

```

PROCEDURE RESULT IS
BEGIN
    IF TEST_STATUS = PASS THEN
        PUT_MSG ("—— " & TEST_NAME &
            " PASSED -----.");
    ELSE PUT_MSG ("**** " & TEST_NAME &
        " FAILED -----.");
    END IF;
    TEST_STATUS := FAIL;
    TEST_NAME := "NO_NAME";
END RESULT;

```

```

FUNCTION IDENT_INT (X : INTEGER) RETURN INTEGER IS
BEGIN
    IF EQUAL (X, X) THEN
        RETURN X;
    END IF;
    RETURN 0;
END IDENT_INT;

```

— ALWAYS EQUAL.
 — ALWAYS EXECUTED.
 — NEVER EXECUTED.

```

FUNCTION IDENT_CHAR (X : CHARACTER) RETURN CHARACTER IS
BEGIN
    IF EQUAL (CHARACTER'POS(X), CHARACTER'POS(X)) THEN
        RETURN X;
    END IF;
    RETURN '0';
END IDENT_CHAR;

```

— ALWAYS
 — EQUAL.
 — ALWAYS EXECUTED.
 — NEVER EXECUTED.

```

FUNCTION IDENT_BOOL (X : BOOLEAN) RETURN BOOLEAN IS
BEGIN
  IF EQUAL (BOOLEAN'POS(X), BOOLEAN'POS(X)) THEN    --ALWAYS
    RETURN X;                                         -- EQUAL.
    -- ALWAYS EXECUTED.
  END IF;
  RETURN FALSE;
END IDENT_BOOL;

FUNCTION IDENT_STR (X : STRING) RETURN STRING IS
BEGIN
  IF EQUAL (X'LENGTH, X'LENGTH) THEN    -- ALWAYS EQUAL.
    RETURN X;                           -- ALWAYS EXECUTED.
  END IF;
  RETURN "";
END IDENT_STR;

FUNCTION EQUAL (X, Y : INTEGER) RETURN BOOLEAN IS
  REC_LIMIT : CONSTANT INTEGER RANGE 1..100 := 3; -- RECURSION
  Z : BOOLEAN;                                   -- LIMIT.
  -- RESULT.
BEGIN
  IF X < 0 THEN
    IF Y < 0 THEN
      Z := EQUAL (-X, -Y);
    ELSE Z := FALSE;
    END IF;
  ELSIF X > REC_LIMIT THEN
    Z := EQUAL (REC_LIMIT, Y-X+REC_LIMIT);
  ELSIF X > 0 THEN
    Z := EQUAL (X-1, Y-1);
  ELSE Z := Y = 0;
  END IF;
  RETURN Z;
EXCEPTION
  WHEN OTHERS =>
    RETURN X = Y;
END EQUAL;

BEGIN
  NULL;
END REPORT;

```

Appendix C

BNF

This appendix contains the BNF used by Garlington's compiler. Terminals will be represented by uppercase letters. Nonterminals will be represented by lower case letters.

1. system goal symbol ::= END program END
2. program ::=
3. program ::= compl_unit_list
4. compl_unit_list ::= compilation_unit ;
5. compl_unit_list ::= compl_unit_list compilation_unit ;
6. vertical_bar ::= |
7. vertical_bar ::= !
8. id ::= *IDENTIFIER*
9. char ::= *CHAR*
10. string ::= *STRING*
11. number ::= *NUMBER*
12. enumeration_literal ::= id
13. enumeration_literal ::= char
14. pragma_list_option ::=
15. pragma_list_option ::= pragma_list
16. pragma_list ::= pragma ;
17. pragma_list ::= pragma_list pragma ;
18. pragma ::= PRAGMA id argument_list_option
19. argument_list_option ::=
20. argument_list_option ::= (argument_list)
21. argument_list ::= argument
22. argument_list ::= argument_list , argument
23. argument ::= expression
24. argument ::= id => expression
25. declarative_part ::=
26. declarative_part ::= declaration_list
27. declaration_list ::= declarative_item ;

28. declaration_list ::= declaration_list declarative_item ;
 29. declarative_item ::= object_declaration
 30. declarative_item ::= number_declaration
 31. declarative_item ::= use_clause
 32. declarative_item ::= type_declaration
 33. declarative_item ::= subtype_declaration
 34. declarative_item ::= exception_declaration
 35. declarative_item ::= renaming_declaration
 36. declarative_item ::= pragma
 37. declarative_item ::= package_declaration
 38. declarative_item ::= subprogram_declaration
 39. declarative_item ::= task_declaration
 40. declarative_item ::= representation_specification
 41. object_declaration ::= identifier_list : constant_option
 subtype_indication initialization_option
 42. object_declaration ::= id : constant_option
 subtype_indication initialization_option
 43. object_declaration ::= identifier_list : constant_option
 array_type_definition initialization_option
 44. object_declaration ::= id : constant_option
 array_type_definition initialization_option
 45. number_declaration ::= identifier_list : CONSTANT
 becomes expression
 46. number_declaration ::= id : CONSTANT becomes expression
 47. becomes ::= :=
 48. constant_option ::=
 49. constant_option ::= CONSTANT
 50. initialization_option ::=
 51. initialization_option ::= becomes expression


```

77. range ::= simple_expression .. simple_expression
78. enumeration_type_definition ::=
      ( enumeration_literal_list )
79. enumeration_literal_list ::= enumeration_literal
80. enumeration_literal_list ::= enumeration_literal_list ,
      enumeration_literal
81. accuracy_constraint ::= DIGITS simple_expression
      range_constraint_option
82. accuracy_constraint ::= DELTA simple_expression
      range_constraint_option
83. array_type_definition ::= ARRAY ( index_list ) OF
      subtype_indication
84. index_list ::= index
85. index_list ::= index_list , index
86. index ::= name RANGE <>
87. index ::= full_discrete_range
88. index ::= name
89. discrete_range ::= name range_constraint_option
90. discrete_range ::= range
91. full_discrete_range ::= name range_constraint
92. full_discrete_range ::= range
93. component_association ::= choice_list => expression
94. component_association ::= name accuracy_constraint
95. choice_list ::= choice
96. choice_list ::= choice_list vertical_bar choice
97. choice ::= simple_expression
98. choice ::= full_discrete_range
99. choice ::= OTHERS
100. record_type_definition ::= RECORD component_list
      END RECORD

```

```

101. component_list ::=
102. component_list ::= compon_decl_list variant_part_option
103. component_list ::= variant_part
104. component_list ::= NULL ;
105. compon_decl_list ::= compon_decl
106. compon_decl_list ::= compon_decl_list compon_decl
107. compon_decl ::= identifier_list : subtype_indication
                    initialization_option ;
108. compon_decl ::= id : subtype_indication
                    initialization_option ;
109. compon_decl ::= identifier_list : array_type_definition
                    initialization_option ;
110. compon_decl ::= id : array_type_definition
                    initialization_option ;
111. variant_part_option ::=
112. variant_part_option ::= variant_part
113. variant_part ::= CASE name IS variant_list_option
                    END CASE ;
114. variant_list_option ::=
115. variant_list_option ::= variant_list
116. variant_list ::= variant
117. variant_list ::= variant_list variant
118. variant ::= WHEN choice_list => component_list
119. access_type_definition ::= ACCESS subtype_indication
120. id_option ::=
121. id_option ::= id
122. identifier_list ::= id , id
123. identifier_list ::= identifier_list , id
124. name_list ::= name
125. name_list ::= name_list , name

```

126. name ::= id
127. name ::= indexed_component
128. name ::= selected_component
129. name ::= attribute
130. indexed_component ::= name generalized_expression_list
131. indexed_component ::= name ()
132. selected_component ::= name . id
133. selected_component ::= name . ALL
134. selected_component ::= name . string
135. attribute ::= name ' id
136. attribute ::= name ' DIGITS
137. attribute ::= name ' DELTA
138. attribute ::= name ' RANGE
139. subprogram_name ::= name
140. subprogram_name ::= string
141. literal ::= string
142. literal ::= number
143. literal ::= char
144. literal ::= NULL
145. expression_option ::=
146. expression_option ::= expression
147. generalized_expression_list ::= gel_head)
148. gel_head ::= l_paren generalized_expression
149. gel_head ::= gel_head , generalized_expression
150. l_paren ::= (
151. generalized_expression ::= expression

- ```

152. generalized_expression ::= simple_expression ..
 simple_expression
153. generalized_expression ::= component_association
154. generalized_expression ::= name range_constraint
155. expression ::= and_expression
156. expression ::= or_expression
157. expression ::= xor_expression
158. expression ::= and_then_expression
159. expression ::= or_else_expression
160. expression ::= relation
161. and_expression ::= relation AND relation
162. and_expression ::= and_expression AND relation
163. or_expression ::= relation OR relation
164. or_expression ::= or_expression OR relation
165. xor_expression ::= relation XOR relation
166. xor_expression ::= xor_expression XOR relation
167. and_then_expression ::= relation and_then relation
168. and_then_expression ::= and_then_expression or_else
 relation
169. and_then ::= AND THEN
170. or_else_expression ::= relation or_else relation
171. or_else_expression ::= or_else_expression or_else
 relation
172. or_else ::= OR ELSE
173. relation ::= simple_expression
174. relation ::= simple_expression = simple_expression
175. relation ::= simple_expression /= simple_expression
176. relation ::= simple_expression < simple_expression
177. relation ::= simple_expression <= simple_expression

```

```

178. relation ::= simple_expression > simple_expression
179. relation ::= simple_expression >= simple_expression
180. relation ::= simple_expression IN subtype_indication
181. relation ::= simple_expression IN range
182. relation ::= simple_expression NOT IN
 subtype_indication
183. relation ::= simple_expression NOT IN range
184. unop_term ::= + term
185. unop_term ::= - term
186. unop_term ::= NOT term
187. simple_expression ::= simple_expression + term
188. simple_expression ::= simple_expression - term
189. simple_expression ::= simple_expression & term
190. simple_expression ::= term
191. simple_expression ::= unop_term
192. term ::= term * factor
193. term ::= term / factor
194. term ::= term MOD factor
195. term ::= term REM factor
196. term ::= factor
197. factor ::= primary
198. factor ::= primary ** primary
199. primary ::= literal
200. primary ::= name
201. primary ::= allocator
202. primary ::= name ' generalized_expression_list
203. primary ::= generalized_expression_list

```

```

204. allocator ::= NEW name
205. statement_list ::= statement ;
206. statement_list ::= statement_list statement ;
207. statement ::= opt_label_list unlabelled_statement
208. statement ::= pragma
209. opt_label_list ::=
210. opt_label_list ::= label_list
211. label_list ::= label
212. label_list ::= label_list label
213. unlabelled_statement ::= simple_statement
214. unlabelled_statement ::= compound_statement
215. simple_statement ::= assignment_statement
216. simple_statement ::= name
217. simple_statement ::= exit_statement
218. simple_statement ::= return_statement
219. simple_statement ::= goto_statement
220. simple_statement ::= raise_statement
221. simple_statement ::= abort_statement
222. simple_statement ::= delay_statement
223. simple_statement ::= name ' generalized_expression_list
224. simple_statement ::= NULL
225. compound_statement ::= if_statement END IF
226. compound_statement ::= case_statement END CASE
227. compound_statement ::= accept_statement
228. compound_statement ::= select_statement END SELECT
229. compound_statement ::= loop_statement END LOOP
 id_option
230. compound_statement ::= block END id_option

```

231. tag\_option ::=   
 232. tag\_option ::= id :   
 233. label ::= << id >>   
 234. assignment\_statement ::= name becomes expression   
 235. if\_statement ::= if\_head   
 236. if\_statement ::= if\_head\_else statement\_list   
 237. if\_head ::= if\_condition THEN statement\_list   
 238. if\_head ::= if\_head\_elsif\_condition THEN statement\_list   
 239. if\_condition ::= IF condition   
 240. if\_head\_elsif\_condition ::= if\_head\_elsif condition   
 241. if\_head\_elsif ::= if\_head ELSIF   
 242. if\_head\_else ::= if\_head ELSE   
 243. condition ::= expression   
 244. case\_statement ::= case\_header alternative\_list\_option   
 245. case\_header ::= CASE expression IS   
 246. alternative\_list\_option ::=   
 247. alternative\_list\_option ::= alternative\_list   
 248. alternative\_list ::= alternative   
 249. alternative\_list ::= alternative\_list alternative   
 250. alternative ::= pre\_alternative statement\_list   
 251. pre\_alternative ::= WHEN choice\_list =>   
 252. loop\_statement ::= tag\_option loop\_intro statement\_list   
 253. loop\_intro ::= LOOP   
 254. loop\_intro ::= WHILE condition LOOP   
 255. loop\_intro ::= FOR id IN reverse\_option  
                                   discrete\_range LOOP   
 256. reverse\_option ::=



283. subp\_spec\_is\_dp ::= subp\_spec\_is declarative\_part  
 284. designator\_option ::=   
 285. designator\_option ::= designator  
 286. designator ::= id  
 287. designator ::= string  
 288. return\_option ::=   
 289. return\_option ::= RETURN subtype\_indication  
 290. formal\_part\_option ::=   
 291. formal\_part\_option ::= ( parameter\_declaration\_list )  
 292. parameter\_declaration\_list ::= parameter\_declaration  
 293. parameter\_declaration\_list ::=   
     parameter\_declaration\_list : parameter\_declaration  
 294. parameter\_declaration ::= identifier\_list : mode\_option  
     subtype\_indication initialization\_option  
 295. parameter\_declaration ::= id : mode\_option  
     subtype\_indication initialization\_option  
 296. mode\_option ::=   
 297. mode\_option ::= IN  
 298. mode\_option ::= OUT  
 299. mode\_option ::= IN OUT  
 300. package\_declaration ::= pkg\_spec\_dcl  
 301. package\_declaration ::= gen\_inst\_pkg  
 302. package\_declaration ::= PACKAGE BODY id IS SEPARATE  
 303. package\_declaration ::= pkg\_body\_dcl  
 304. gen\_inst\_pkg ::= pkg\_spec\_hdr IS NEW name  
 305. pkg\_spec\_dcl ::= pkg\_spec\_hdr\_is\_dp  
     private\_decl\_part\_option END id\_option  
 306. pkg\_spec\_hdr\_is\_dp ::= pkg\_spec\_hdr\_is declarative\_part  
 307. pkg\_spec\_hdr ::= pkg\_spec\_hdr IS



```

333. opt_type_kw ::=
334. opt_type_kw ::= TYPE
335. opt_task_is ::=
336. opt_task_is ::= IS opt_entries rep_spec_list_option
 END id_option
337. rep_spec_list_option ::=
338. rep_spec_list_option ::= rep_spec_list
339. rep_spec_list ::= representation_specification ;
340. rep_spec_list ::= rep_spec_list
 representation_specification ;
341. opt_entries ::=
342. opt_entries ::= entry_dcl_list
343. entry_dcl_list ::= entry_declaration ;
344. entry_dcl_list ::= entry_dcl_list entry_declaration ;
345. synchronization_statement ::= accept_statement
346. synchronization_statement ::= delay_statement
347. entry_declaration ::= ENTRY id (discrete_range)
 formal_part_option
348. entry_declaration ::= ENTRY id formal_part_option
349. accept_hdr ::= ACCEPT id formal_part_option
350. accept_hdr ::= ACCEPT paren_name formal_part_option
351. accept_statement ::= accept_hdr
352. accept_statement ::= accept_hdr_do statement_list END
 id_option
353. accept_hdr_do ::= accept_hdr DO
354. paren_name ::= id (expression)
355. delay_statement ::= DELAY simple_expression
356. select_statement ::= select_kw select_body
357. select_statement ::= select_kw conditional_entry_call

```

```

358. select_statement ::= select_kw timed_entry_call
359. conditional_entry_call ::= entry_list else_kw
 statement_list
360. else_kw ::= ELSE
361. timed_entry_call ::= entry_list_or_del stmt_list_option
362. entry_list_or_del ::= entry_list OR delay_statement ;
363. entry_list ::= entry_call_statement stmt_list_option
364. entry_call_statement ::= name ;
365. stmt_list_option ::=
366. stmt_list_option ::= statement_list
367. select_kw ::= SELECT
368. select_body ::= select_alternative_list else_kw
 statement_list
369. select_body ::= select_alternative_list
370. select_alternative_list ::= select_alternative
371. select_alternative_list ::= select_alternative_list OR
 select_alternative
372. select_alt_front ::= condition_option
 synchronization_option ;
373. select_alternative ::= select_alt_front statement_list
374. select_alternative ::= select_alt_front
375. select_alternative ::= condition_option TERMINATE ;
376. condition_option ::=
377. condition_option ::= WHEN condition =>
378. abort_statement ::= ABORT name_list
379. comp_unithdr ::= pragma_list_option context_list_option
380. compilation_unit ::= comp_unithdr l_unit
381. compilation_unit ::= comp_unithdr SEPARATE (
 designator_dot_name) s_unit

```

```

382. designator_dot_name ::= designator
383. designator_dot_name ::= designator_dot_name .
 designator
384. s_unit ::= c_body
385. s_unit ::= task_body
386. c_body ::= subp_body_dcl
387. c_body ::= pkg_body_dcl
388. l_unit ::= c_body
389. l_unit ::= subp_spec
390. l_unit ::= pkg_spec_dcl
391. l_unit ::= gen_inst_pkg
392. l_unit ::= gen_inst_subp
393. context_list_option ::=
394. context_list_option ::= context_list
395. context_list ::= context
396. context_list ::= context_list context
397. context ::= with_clause
398. context ::= with_clause use_clause ;
399. with_clause ::= WITH name_list ;
400. exception_declaration ::= identifier_list : EXCEPTION
401. exception_declaration ::= id : EXCEPTION
402. exception_handler_list_option ::=
403. exception_handler_list_option ::=
 exception_handler_list
404. exception_handler_list ::= exception_handler
405. exception_handler_list ::= exception_handler_list
 exception_handler
406. exception_handler ::= eh_pre_stm statement_list

```

```

407. eh_pre_stm ::= WHEN exception_choice_list =>
408. exception_choice_list ::= exception_choice
409. exception_choice_list ::= exception_choice_list
 vertical_bar exception_choice
410. exception_choice ::= name
411. exception_choice ::= OTHERS
412. raise_statement ::= RAISE name
413. raise_statement ::= RAISE
414. generic_formal_list_option ::=
415. generic_formal_list_option ::= generic_formal_list
416. generic_part ::= GENERIC generic_formal_list_option
417. generic_formal_list ::= generic_formal ;
418. generic_formal_list ::= generic_formal_list
 generic_formal ;

419. generic_formal ::= parameter_declaration
420. generic_formal ::= TYPE id discriminant_part_option
 IS generic_type_definition
421. generic_formal ::= WITH subp_hdr
422. generic_formal ::= WITH subp_hdr IS subprogram_name
423. generic_formal ::= WITH subp_hdr IS <>
424. generic_type_definition ::= (<>)
425. generic_type_definition ::= RANGE <>
426. generic_type_definition ::= DELTA <>
427. generic_type_definition ::= DIGITS <>
428. generic_type_definition ::= array_type_definition
429. generic_type_definition ::= access_type_definition
430. generic_type_definition ::= private_type_definition
431. representation_specification ::=
 length_spec_or_enum_rep

```

```

432. representation_specification ::=
 record_type_representation
433. representation_specification ::= address_specification
434. length_spec_or_enum_rep ::= FOR name USE expression
435. record_type_representation ::= for_name_use_record
 alignment_clause_option comp_name_loc_list_option
 END RECORD
436. for_name_use_record ::= FOR name USE RECORD
437. alignment_clause_option ::=
438. alignment_clause_option ::= AT MOD simple_expression ;
439. comp_name_loc_list_option ::=
440. comp_name_loc_list_option ::= comp_name_loc_list
441. comp_name_loc_list ::= comp_name_loc ;
442. comp_name_loc_list ::= comp_name_loc_list
 comp_name_loc ;
443. comp_name_loc ::= name AT simple_expression
 range_constraint
444. address_specification ::= FOR name USE AT
 simple_expression

```

## Appendix D

### Removal of Language-features from Valid Tests

The purpose of this appendix is to show that the tests found in the ACVC can still be useful after language-features have been removed from them. It provides an example of a Class A test and shows what it would look like after some features have been removed. Figure D-1 shows the test to be evaluated.

The first case studied is the impact the removal of record structures would have on the test. Figure D-2 illustrates what the test would look like after all uses of the record structure were removed. The only test objective not accomplished by the test is testing the acceptance of '.'. The remainder of the test is still valid.

If the array is added to the list of unsupported features the resulting test would look like the test shown in figure D-3. The use of ':', '(', ')', '|', ',' and ''' would no longer be tested, but the remaining test would be valid.

This process can continue until all that remains is the procedure calls to TEST and RESULT, provided they were recoded using the supported language features. When the string type is no longer supported the TEST procedure call would be an illegal construct.

```

WITH REPORT;
PROCEDURE A21001A IS
 USE REPORT;

BEGIN
 TEST ("A21001A", "CHECK THAT BASIC CHARACTER SET IS ACCEPTED");

 DECLARE

 TYPE TABLE IS ARRAY (1..10) OF INTEGER;
 A : TABLE := (2 | 4 | 10 => 1 , 1 | 3 | 5..9 => 0) ;
 -- USE OF : () | ,

 TYPE BUFFER IS
 RECORD
 LENGTH : INTEGER;
 POS : INTEGER;
 IMAGE : INTEGER;
 END RECORD; -- USED TO TEST . LATER

 R1 : BUFFER;

 ABCDEFGHIJKLM : INTEGER; -- USE OF ABCDEFGHIJKLM
 NOPQRSTUVWXYZ : INTEGER; -- USE OF NOPQRSTUVWXYZ
 Z_1234567890 : INTEGER; -- USE OF _1234567890
 I1, I2, I3 : INTEGER;

 C1, C2 : STRING (1..6);
 C3 : STRING (1..12);

 BEGIN
 I1 := 2 * (3 - 1 + 2) / 2 ; I2 := 8 ; -- USES () * + - / ;
 C1 := "ABCDEF" ; -- USE OF "
 C2 := C1;
 C3 := C1 & C2 ; -- USE OF &
 I2 := 16#D#; -- USE OF #
 I3 := A'LAST; -- USE OF '
 R1.POS := 3; -- USE OF .
 IF I1 > 2 AND
 I1 = 4 AND
 I1 < 8 THEN -- USE OF > = <
 NULL;
 END IF;

 END;
 RESULT;
END A21001A;

```

Figure D-1. Class A Test

```

WITH REPORT;
PROCEDURE A21001A IS
 USE REPORT;

BEGIN

 TEST ("A21001A", "CHECK THAT BASIC CHARACTER SET IS ACCEPTED");

 DECLARE

 TYPE TABLE IS ARRAY (1..10) OF INTEGER;
 A : TABLE := (2 | 4 | 10 => 1 , 1 | 3 | 5..9 => 0) ;
 — USE OF : () | ,

 ABCDEFGHIJKLM : INTEGER; — USE OF ABCDEFGHIJKLM
 NOPQRSTUVWXYZ : INTEGER; — USE OF NOPQRSTUVWXYZ
 Z_1234567890 : INTEGER; — USE OF _1234567890
 I1, I2, I3 : INTEGER;

 C1, C2 : STRING (1..6);
 C3 : STRING (1..12);

 BEGIN

 I1 := 2 * (3 - 1 + 2) / 2 ; I2 := 8.; — USES () * + - / ;
 C1 := "ABCDEF" ; — USE OF "
 C2 := C1;
 C3 := C1 & C2 ; — USE OF &
 I2 := 16#D#; — USE OF #
 I3 := A'LAST; — USE OF '
 IF I1 > 2 AND
 I1 = 4 AND
 I1 < 8 THEN — USE OF > = <
 NULL;
 END IF;

 END;
 RESULT;
END A21001A;

```

Figure D-2. Class A Test with record removed

```

WITH REPORT;
PROCEDURE A21001A IS
 USE REPORT;

BEGIN
 TEST ("A21001A", "CHECK THAT BASIC CHARACTER SET IS ACCEPTED");

 DECLARE
 ABCDEFGHIJKLM : INTEGER; -- USE OF ABCDEFGHIJKLM
 NOPQRSTUVWXYZ : INTEGER; -- USE OF NOPQRSTUVWXYZ
 Z_1234567890 : INTEGER; -- USE OF _1234567890
 I1, I2, I3 : INTEGER;
 C1, C2 : STRING (1..6);
 C3 : STRING (1..12);

 BEGIN
 I1 := 2 * (3 - 1 + 2) / 2 ; I2 := 8 ; -- USES () * + - / ;
 C1 := "ABCDEF" ; -- USE OF "
 C2 := C1;
 C3 := C1 & C2 ; -- USE OF &
 I2 := 16#D#; -- USE OF #
 IF I1 > 2 AND
 I1 = 4 AND
 I1 < 8 THEN -- USE OF > = <
 NULL;
 END IF;
 END;
 RESULT;
END A21001A;

```

Figure D-3. Class A Test with records and arrays removed

Class A tests in most cases still accomplish many of the test objectives even when language-features are removed. In most cases, Class C and Class D tests do not accomplish their test objectives when features are removed. The prime purpose for removing features from Class C and Class D tests is so they will not fail compilation. If they failed compilation, they would require manual analysis to determine they failed because they used unsupported features.

AD-A127 333

PRELIMINARY DESIGN AND IMPLEMENTATION OF A METHOD FOR  
VALIDATING EVOLVING ADA COMPILERS(U) AIR FORCE INST OF  
TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI..

12

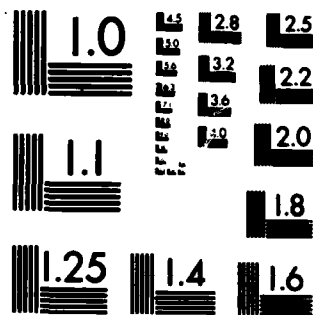
UNCLASSIFIED

E D MILLER MAR 83 AFIT/GCS/MA/83M-1

F/G 9/2

NL

END  
DATE  
FILMED  
583  
DTIC



MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A

## APPENDIX E

### Manual Evaluation of Tests

This appendix was included to show the manual effort required to analyze tests contained in the ACVC. A small segment of the test shown in figure 2-3 is analyzed using the BNF contained in the LRM (Ref 1). The segment analyzed is shown in figure E-1.

```
WITH REPORT;
PROCEDURE A21001A IS
 USE REPORT;

BEGIN
 TEST ("A21001A", "CHECK THAT BASIC CHARACTER SET IS ACCEPTED");

 DECLARE

 TYPE TABLE IS ARRAY (1..10) OF INTEGER;
 A : TABLE := (2 | 4 | 10 => 1 , 1 | 3 | 5..9 => 0) ;
 -- USE OF : () | ,

END;
```

Figure E-1. Test Segment Analyzed Manually

In the following analysis, the code being analyzed is found between the asterisks.

TEST A21001A.ADA

\*\*\*\*\* WITH REPORT; \*\*\*\*\*

compilation ::= compilation\_unit

compilation\_unit ::= context\_specification  
                                subprogram\_body

context\_specification ::= with\_clause

with\_clause ::= WITH unit\_name;

name ::= identifier

identifier ::= letter {letter\_or\_digit}

letter ::= upper\_case\_letter

\*\*\*\*\* PROCEDURE A21001A IS \*\*\*\*\*

subprogram\_body ::= subprogram\_specification IS  
                                declarative\_part  
                                BEGIN  
                                        sequence\_of\_statements  
                                END [designator];

subprogram\_specification ::= PROCEDURE identifier

identifier ::= letter {letter\_or\_digit}

letter ::= upper\_case\_letter

\*\*\*\*\* USE REPORT ; \*\*\*\*\*

declarative\_part ::= declarative\_item

declarative\_item ::= use\_clause

use\_clause ::= USE package\_name

name ::= identifier

identifier ::= letter {letter\_or\_digit}

letter ::= upper\_case\_letter

\*\*\*\*\* BEGIN \*\*\*\*\*

sequence\_of\_statements ::= statement { statement }

\*\*\*\*\*  
TEST("A21001A","CHECK THAT BASIC CHARACTER SET IS ACCEPTED");  
\*\*\*\*\*

\*\*\*\*\* TEST \*\*\*\*\*

statement ::= simple\_statement

simple\_statement ::= procedure\_call

procedure\_call ::= procedure\_name [actual\_parameter\_part];

name ::= identifier

identifier ::= letter { letter\_or\_digit }

letter ::= upper\_case\_letter

actual\_parameter\_part ::= (parameter\_association  
                                  { ,parameter\_association })

\*\*\*\*\* "A21001A" \*\*\*\*\*

parameter\_association ::= actual\_parameter

actual\_parameter ::= expression

expression ::= relation

relation ::= simple\_expression

simple\_expression ::= term

term ::= factor

factor ::= primary

primary ::= literal

literal ::= character\_string

character\_string ::= "{character}"

\*\*\*\*\* "CHECK THAT BASIC CHARACTER SET IS ACCEPTED"\*\*\*\*\*

parameter\_association ::= actual\_parameter

actual\_parameter ::= expression

expression ::= relation

relation ::= simple\_expression

simple\_expression ::= term

term ::= factor

factor ::= primary

primary ::= literal

literal ::= character\_string

character\_string ::= "{character}"

\*\*\*\*\* DECLARE \*\*\*\*\*

statement ::= compound\_statement

compound\_statement ::= block

block ::= DECLARE  
    declarative\_part  
    BEGIN  
        sequence\_of\_statements  
    END [block\_identifier]

declarative\_part ::= {declarative\_item}

\*\*\*\*\* TYPE TABLE IS \*\*\*\*\*

declarative\_item ::= declaration

declaration ::= type\_declaration

type\_declaration ::= TYPE identifier IS type\_definition;

identifier ::= letter {letter\_or\_digit}

letter ::= upper\_case\_letter

\*\*\*\*\* ARRAY (1..10) OF INTEGER \*\*\*\*\*

type\_definition ::= array\_type\_definition

array\_type\_definition ::= ARRAY index\_constraint OF  
component\_subtype\_indication

index\_constraint ::= (discrete\_range)

discrete\_range ::= range

range ::= simple\_expression..simple\_expression

\*\*\*\*\* 1 \*\*\*\*\*

simple\_expression ::= term

term ::= factor

factor ::= primary .

primary ::= literal

literal ::= numeric\_literal

numeric\_literal ::= decimal\_number

decimal\_number ::= integer

integer ::= digit {digit}

\*\*\*\*\* 10 \*\*\*\*\*

simple\_expression ::= term

term ::= factor

factor ::= primary

primary ::= literal

literal ::= numeric\_literal

numeric\_literal ::= decimal\_number

decimal\_number ::= integer

integer ::= digit {digit}

\*\*\*\*\* INTEGER \*\*\*\*\*

subtype\_indication ::= type\_mark

type\_mark ::= type\_name

name ::= identifier

identifier ::= letter {letter\_or\_digit}

letter ::= upper\_case\_letter

\*\*\*\* A : TABLE := (2 | 4 | 10 => 1, 1 | 3 | 5..9 => 0); \*\*\*\*

declarative\_item ::= declaration

declaration ::= object\_declaration

object\_declaration ::= identifier\_list:  
                          subtype\_indication [:= expression]

\*\*\*\*\* A \*\*\*\*\*

identifier\_list ::= identifier

identifier ::= letter {letter\_or\_digit}

letter ::= upper\_case\_letter

\*\*\*\*\* TABLE \*\*\*\*\*

subtype\_indication ::= type\_mark

type\_mark ::= type\_name

name ::= identifier

identifier ::= letter {letter\_or\_digit}

letter ::= upper\_case\_letter

\*\*\*\*\* (2|4|10 => 1, 1|3|5..9 => 0) \*\*\*\*\*

expression ::= relation

relation ::= simple\_expression

simple\_expression ::= term

term ::= factor

```

factor ::= primary
primary ::= (expression)
expression ::= relation
relation ::= simple_expression
simple_expression ::= term
term ::= factor
factor ::= primary
primary ::= aggregate
aggregate ::= (component_association { , component_association })

***** 2 | 4 | 10 => 1 *****
component_association ::= [choice { | choice } =>] expression

***** 2 *****

choice ::= simple_expression
simple_expression ::= term
term ::= factor
factor ::= primary
primary ::= literal
literal ::= numeric_literal
numeric_literal ::= decimal_number
decimal_number ::= integer
integer ::= digit { digit }

***** 4 *****

choice ::= simple_expression
simple_expression ::= term
term ::= factor
factor ::= primary

```

primary ::= literal

literal ::= numeric\_literal

numeric\_literal ::= decimal\_number

decimal\_number ::= integer [integer] [exponent]

integer ::= digit { [underscore] digit }

\*\*\*\*\* 10 \*\*\*\*\*

choice ::= simple\_expression

simple\_expression ::= term

term ::= factor

factor ::= primary

primary ::= literal

literal ::= numeric\_literal

numeric\_literal ::= decimal\_number

decimal\_number ::= integer

integer ::= digit { digit }

\*\*\*\*\* 1 \*\*\*\*\*

expression ::= relation

relation ::= simple\_expression

simple\_expression ::= term

term ::= factor

factor ::= primary

primary ::= literal

literal ::= numeric\_literal

numeric\_literal ::= decimal\_number

decimal\_number ::= integer

integer ::= digit { digit }

\*\*\*\*\* 1 | 3 | 5..9 => 0 \*\*\*\*\*

component\_association ::= [choice { | choice } =>] expression

\*\*\*\*\* 1 \*\*\*\*\*

choice ::= simple\_expression

simple\_expression ::= term

term ::= factor

factor ::= primary

primary ::= literal

factor ::= primary

primary ::= literal

literal ::= numeric\_literal

numeric\_literal ::= decimal\_number

decimal\_number ::= integer

integer ::= digit { digit }

\*\*\*\*\* 3 \*\*\*\*\*

choice ::= simple\_expression

simple\_expression ::= term

term ::= factor

factor ::= primary

primary ::= literal

numeric\_literal ::= decimal\_number

decimal\_number ::= integer

integer ::= digit { digit }

\*\*\*\*\* 5.9 \*\*\*\*\*

choice ::= discrete\_range

discrete\_range ::= range

range ::= simple\_expression..simple\_expression

\*\*\*\*\* 5 \*\*\*\*\*

simple\_expression ::= term

term ::= factor

factor ::= primary

primary ::= literal

numeric\_literal ::= decimal\_number

decimal\_number ::= integer

integer ::= digit {digit}

\*\*\*\*\* 9 \*\*\*\*\*

simple\_expression ::= term

term ::= factor

factor ::= primary

primary ::= literal

numeric\_literal ::= decimal\_number

decimal\_number ::= integer

integer ::= digit {digit}

\*\*\*\*\* 0 \*\*\*\*\*

expression ::= relation

relation ::= simple\_expression

simple\_expression ::= term

term ::= factor

factor ::= primary

primary ::= literal  
literal ::= numeric\_literal  
numeric\_literal ::= decimal\_number  
decimal\_number ::= integer  
integer ::= digit {digit}

## APPENDIX F

### Description of the Garlington Compiler

This appendix describes the Ada compiler used to parse the test programs. It was developed by Alan Garlington as part of his thesis effort in the design and implementation of an Ada pseudo-machine (Ref 5).

The portion of the user's guide which describes the features implemented is reproduced here for convenience.

1. Integer Variables. Number declarations and variable initializations are not implemented.
2. Package declarations.
3. Procedures and functions with parameters (mode types may be specified).
4. Task declarations.
5. Selected components may be used to open visibility to objects that are within scope but which are not directly visible.
6. Most integer arithmetic or Boolean expressions may be used including those using short circuit conditions. However, the following list of operators has not been implemented: REM, \*\*, &, IN.
7. The following statements may be used:
  - a. Assignment
  - b. Procedure, function or entry calls
  - c. Exit
  - d. Return
  - e. IF THEN ELSIF ELSE
  - f. Accept
  - g. Loops (except FOR loop)

The aspect of the Garlington compiler which had the most significant influence on this project was the parser. The parser used by Garlington was a LR (1) parsing automaton. It is a bottom-up, finite-state machine whose operations are directed by a set of language-specific tables. These tables were generated using the LR package from Lawrence Livermore Laboratory (Ref 8). The parser used was designed to parse the full Ada language as described in the 1980 version of the "Reference Manual for the Ada Programming Language".

## APPENDIX G

### Modifications to the Garlington Compiler

This appendix describes the changes made in the Garlington compiler in order to get it to compile on the VAX 11-780. There were four primary changes made to the Garlington compiler.

The first change required was a case conversion. The version taken off the DEC-10 was written in all upper-case letters. To compile on the VAX, all reserved words must be in lower-case. The program was run through a case conversion routine to change it to lower case.

The second change removed all line numbers from the program. This was accomplished using a program to filter out line numbers.

The third change removed all CYBER and DEC-10 specific routines. This included the date and time facilities used by the compiler.

The final and probably most serious modification was related to the structure of the case statements used in the program. The version of Pascal implemented on the DEC-10 supported the use an OTHERS form to catch anything that fell through the case statement. The version implemented on the VAX does not support the OTHERS statement. This required inserting a range check before each case statement.

## APPENDIX H

### Empty Productions

This appendix contains a list of all the empty and potentially empty productions found in the BNF contained in Appendix C. These productions are listed by their corresponding number.

|    |     |     |     |     |     |
|----|-----|-----|-----|-----|-----|
| 2  | 54  | 120 | 259 | 296 | 341 |
| 3  | 55  | 121 | 260 | 297 | 342 |
| 14 | 74  | 145 | 263 | 298 | 365 |
| 15 | 75  | 146 | 264 | 299 | 366 |
| 19 | 101 | 209 | 266 | 316 |     |
| 20 | 102 | 210 | 267 | 317 |     |
| 25 | 103 | 231 | 284 | 333 |     |
| 26 | 104 | 232 | 285 | 334 |     |
| 48 | 111 | 246 | 288 | 335 |     |
| 49 | 112 | 247 | 289 | 336 |     |
| 50 | 114 | 256 | 290 | 337 |     |
| 51 | 115 | 257 | 291 | 338 |     |

## APPENDIX I

### Recursive Productions

This appendix contains a list of all the recursive and potentially recursive productions found in the BNF contained in Appendix C. These productions are listed by their corresponding number.

|    |     |     |     |     |     |
|----|-----|-----|-----|-----|-----|
| 4  | 84  | 148 | 188 | 292 | 404 |
| 5  | 85  | 149 | 189 | 293 | 405 |
| 16 | 95  | 161 | 192 | 339 | 408 |
| 17 | 96  | 162 | 193 | 340 | 409 |
| 21 | 105 | 163 | 194 | 343 | 417 |
| 22 | 106 | 164 | 195 | 344 | 418 |
| 27 | 116 | 165 | 205 | 370 | 441 |
| 28 | 117 | 166 | 206 | 371 | 442 |
| 56 | 122 | 167 | 211 | 382 |     |
| 57 | 123 | 168 | 212 | 383 |     |
| 79 | 124 | 170 | 248 | 395 |     |
| 80 | 125 | 171 | 249 | 396 |     |

## VITA

Edward D. Miller, Jr. was born on 1 May 1954 in Middlesboro, Kentucky. He attended Worthington High School in Worthington, Ohio and graduated in 1972. In July of that year, he entered the United States Military Academy in West Point, New York and subsequently graduated with a Bachelor of Science degree in June of 1976. After graduation Captain Miller attended the Airborne Course at Fort Benning, Georgia, the Signal Officer Basic Course at Fort Gordon, Georgia and the Communications and Electronics Staff Officer course at Fort Sill, Oklahoma. He was then assigned as the Communications-Electronics Staff Officer for the 588th Engineer Battalion (Corps) at Fort Polk, Louisiana. While at Fort Polk, he also served as a Platoon Leader and Company Commander in the 5th Signal Battalion, 5th Infantry Division (Mechanized). After leaving Fort Polk, Captain Miller attended the Signal Officers Advanced Course at Fort Gordon, Georgia and the Teleprocessing Operations Course at the Air Force Institute of Technology at Wright-Patterson AFB, Ohio. After completing the Teleprocessing Operations Course, he entered the Air Force Institute of Technology School of Engineering.

Permanent address: 416 Haymore Ave., N.  
Worthington, Ohio 43085

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| REPORT DOCUMENTATION PAGE                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |                                                                | READ INSTRUCTIONS<br>BEFORE COMPLETING FORM |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------|---------------------------------------------|
| 1. REPORT NUMBER<br>AFIT/GCS/MA/83M-1                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           | 2. GOVT ACCESSION NO.<br>AD-A127333                            | 3. REPORT'S CATALOG NUMBER                  |
| 4. TITLE (and Subtitle)<br>PRELIMINARY DESIGN AND IMPLEMENTATION OF A METHOD<br>FOR VALIDATING EVOLVING ADA COMPILERS                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           | 5. TYPE OF REPORT & PERIOD COVERED<br>MS Thesis                |                                             |
| 7. AUTHOR(s)<br>Edward D. Miller, Jr.<br>CPT USA                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                | 6. PERFORMING ORG. REPORT NUMBER                               |                                             |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS<br>Air Force Institute of Technology (AFIT/EN)<br>Wright-Patterson AFB, Ohio 45433                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  | 8. CONTRACT OR GRANT NUMBER(s)                                 |                                             |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>Language Control Branch (ADOL),<br>Computer Operations Division, Aeronautical Sys Div<br>Wright-Patterson AFB, OH 45433                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              | 10. PROGRAM ELEMENT, PROJECT, TASK<br>AREA & WORK UNIT NUMBERS |                                             |
| 14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | 12. REPORT DATE<br>March, 1983                                 |                                             |
|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | 13. NUMBER OF PAGES<br>113                                     |                                             |
|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | 15. SECURITY CLASS. (of this report)<br>Unclassified           |                                             |
| 16. DISTRIBUTION STATEMENT (of this Report)<br><br>Approved for public release; distribution unlimited                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |                                                                |                                             |
| 17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |                                                                |                                             |
| 18. SUPPLEMENTARY NOTES<br><br>Approved for public release; LAW APR 1983<br>LYNN E. WOLAVER<br>Dean for Research and Professional Development<br>Air Force Institute of Technology (AFIT)<br>Wright-Patterson AFB OH 45433                                                                                                                                                                                                                                                                                                                                                                                                                                      |                                                                |                                             |
| 19. KEY WORDS (Continue on reverse side if necessary and identify by block number)<br><br>Ada<br>Compilers<br>Validation                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |                                                                |                                             |
| 20. ABSTRACT (Continue on reverse side if necessary and identify by block number)<br><br>This project consisted of a preliminary design and a partial implementation of a tool which modifies the existing Ada Compiler Validation Capability (ACVC) test set so it can be used to test evolving Ada compilers. The project evaluated the feasibility of repackaging each of test classes found in the ACVC and suggested methods for repackaging the tests. The tool developed uses a table-driven parser which parses the July 1980 proposed standard. It uses output from the parser to generate a representation of a test program. Once the representation |                                                                |                                             |

DD FORM 1473

1 JAN 73

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

Block 20

is developed, unsupported language-features are removed from it. The remaining representation is output as a valid test program.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)