CTRAN: Transforming Scientific Fortran Programs to Unix Based Computing Environments

Qi Chen

Follow this and additional works at: https://scholarworks.wmich.edu/masters_theses

Part of the Computer Sciences Commons

Recommended Citation
Chen, Qi, "CTRAN: Transforming Scientific Fortran Programs to Unix Based Computing Environments" (1990). Master's Theses. 1057.
https://scholarworks.wmich.edu/masters_theses/1057

This Masters Thesis-Open Access is brought to you for free and open access by the Graduate College at ScholarWorks at WMU. It has been accepted for inclusion in Master's Theses by an authorized administrator of ScholarWorks at WMU. For more information, please contact wmu-scholarworks@wmich.edu.
CTRAN: TRANSFORMING SCIENTIFIC FORTRAN PROGRAMS
TO UNIX BASED COMPUTING ENVIRONMENTS

by
Qi Chen

A Thesis
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Master of Science
Department of Computer Science

Western Michigan University
Kalamazoo, Michigan
August 1990
FORTRAN is one of the most popular scientific programming languages. The efficiency, rich libraries, and popularity of FORTRAN support its continued use as a programming language for scientific computing in the near future. However, in current multiprocessor UNIX environments, C is often better supported. The desire to use existing FORTRAN subroutines within such a C environment leads to several options. In this thesis, we introduce a "modern" scientific programming language called CTRAN for this purpose.

We begin with a brief review of the history of scientific programming languages. An introduction of CTRAN is given in Chapter III. In Chapter IV, we present a CTRAN to C translator called ctran2c. A CTRAN manual is included in appendix A. Issues of conflict, other options and possible extensions are also noted. CTRAN was tested on code from the LINPACK library.
ACKNOWLEDGEMENTS

I wish to express much appreciation to my adviser, Professor John Kapenga, for his valuable ideas and time spent on this thesis. I have gained much from his outstanding guidance and continual encouragement. I would like to thank Professor Elise Kapenga for her excellent guidance of my course work and invaluable suggestions on this manuscript. Special thanks are due to Professor Robert Trenary for serving on my committee and offering valuable comments.

I thank my mother for her encouragement and support. Finally, I thank Songlin for all the love and support he has given me.

Qi Chen
INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
CTRAN: Transforming scientific FORTRAN programs to UNIX-based computing environments

Chen, Qi, M.S.
Western Michigan University, 1990
# TABLE OF CONTENTS

ACNOWLEDGEMENTS ....................................................................................... ii  

CHAPTER  
I. INTRODUCTION ...................................................................................... 1  
II. COMPUTER LANGUAGES AND SCIENTIFIC COMPUTING ...... 3  
III. INTRODUCTION TO CTRAN ............................................................. 18  
   3.1 Introduction ................................................................................... 18  
   3.2 CTRAN Language Characteristics ............................................. 20  
      3.2.1 Character Set ..................................................................... 20  
      3.2.2 Names .............................................................................. 21  
      3.2.3 Keywords ........................................................................ 23  
      3.2.4 Statement Structure ....................................................... 24  
         3.2.4.1 Comment Lines ...................................................... 24  
         3.2.4.2 Continuation Lines ................................................ 25  
      3.2.5 CTRAN Statements ........................................................... 25  
      3.2.6 Program Organization ....................................................... 26  
   3.3 Data Types, Data Structures, and Expressions ....................... 27  
      3.3.1 Data Types ........................................................................ 27  
         3.3.1.1 Integer Data ............................................................. 30  
         3.3.1.2 Real Data ................................................................. 30  
         3.3.1.3 Double Precision Data ............................................ 32  
         3.3.1.4 Complex Data ........................................................ 34  
         3.3.1.5 Double Complex Data ............................................ 35  
         3.3.1.6 Logical Data ........................................................... 35
Table of Contents—Continued

3.3.1.7 Character Data ......................................................... 35
3.3.2 Data Structures ............................................................. 36
  3.3.2.1 Variables .............................................................. 37
  3.3.2.2 Character Substrings ............................................ 37
  3.3.2.3 Arrays ................................................................. 38
3.3.3 Expressions ................................................................. 39
3.4 Control Flow Statements .................................................... 42
  3.4.1 Goto Statements ....................................................... 43
  3.4.2 Logical If Statements ............................................... 44
  3.4.3 Block If Statements ................................................. 44
  3.4.4 Arithmetic If Statements ......................................... 46
  3.4.5 Do Statements ......................................................... 47
  3.4.6 For Statements ........................................................ 50
  3.4.7 While Statements ..................................................... 52
  3.4.8 Repeat Statements ................................................... 52
  3.4.9 Exit_loop Statements ............................................... 53
  3.4.10 Blank Statements and Cycle_loop Statements .......... 54
  3.4.11 Following the Loops ............................................... 55
3.5 Program Structure .......................................................... 55
  3.5.1 Main Program .......................................................... 56
  3.5.2 Intrinsic Functions .................................................. 56
  3.5.3 External Procedures ................................................. 57
    3.5.3.1 Kinds of External Procedures ............................... 57
    3.5.3.2 Referencing an External Function...................... 58
LIST OF TABLES

1. Names of Characters in CTRAN ............................................... 22
2. Some Valid and Invalid Cases of Integer Constant ....................... 30
3. Some Valid Cases of Real Constant ........................................... 31
4. Some Invalid Cases of Real Constant ....................................... 32
5. Some Valid Cases of Double Precision Constant ............................ 33
6. Some Invalid Cases of Double Precision Constant ........................ 33
7. Some Valid Cases of Complex Constant ....................................... 34
8. Some Valid Cases of Character Constant .................................... 36
9. Some Invalid Cases of Character Constant .................................. 36
10. A Summary of Operators and Type Conversion ............................. 42
11. C Types Used in Ctran2c ....................................................... 63
CHAPTER I

INTRODUCTION

This thesis explores several possibilities in procedural programming languages, motivated by the dominance of the UNIX operating system in current workstations and distributed memory multicomputers and the dominance of FORTRAN in scientific computing.

If input and output statements and intrinsic functions are removed then most procedural languages are very similar. It is this intersection in which we are interested. Some of the areas this work touches are: (a) the translation of FORTRAN library routines into C, (b) possible changes in FORTRAN to "modernize" it, (c) a test-bed to try FORTRAN extensions, and (d) a test-bed for trying ideas in compiler optimization.

If a little care is taken, a very simple language can be defined which corresponds closely to well written FORTRAN. We call this language CTRAN. By "very simple" we mean that there is a small LEX program which acts as a Lexical analyzer and a small YACC grammar which leads to a parser, which we, in fact, use to produce a CTRAN to C translator, ctran2c.

The goals in (a) and (b) above clearly conflict. In defining CTRAN, FORTRAN and C are in a sense merged:

1. Some features of FORTRAN are dropped entirely, such as embedded blanks in tokens.
2. Some features of FORTRAN are made FORTRAN Compatibility Options, such as several DO loops ending with the same statement. This supports (a) in the list above.

3. The best we can do, if we were not concerned with (a) in the list above, produces what we call Core CTRAN; Core CTRAN contains several features from C and FORTRAN 8X.

In Chapter II, a brief historical review of scientific programming developments, with respect to language, machine architecture and library growth is given. This provides an understanding of current scientific programming language design and development. The Chapter concludes with a list of features we feel a good procedural language should have.

CTRAN is introduced in Chapter III. Following the goals of CTRAN, the structure of CTRAN is introduced and illustrated by examples. Both Core CTRAN and the FORTRAN Compatibility Options are presented. The implementation techniques are also discussed. A CTRAN to C translator is introduced in Chapter IV. Other tools such as f2c and Ratfor are mentioned.

The appendices contain a CTRAN manual, a CTRAN grammar, a lexical analyzer for CTRAN, and a LINPACK FORTRAN procedure with corresponding CTRAN and C versions.
CHAPTER II
COMPUTER LANGUAGES AND SCIENTIFIC COMPUTING

Scientific computing was the first and remains one of the main applications of computers. Most of the early work in programming language development was concerned with handling numerical scientific problems. There are three driving forces in scientific language development: (1) new computer architectures, (2) new programming language design concepts and compiler technology, and (3) new numerical algorithms and libraries. In this chapter, we will briefly track these three threads as they apply to scientific programming. We begin with an important event, the birth of FORTRAN.

The earliest significant document about FORTRAN is one titled "PRELIMINARY REPORT, specifications for the IBM mathematical FORMula TRANslating System, FORTRAN," dated November 10, 1954 (Sammet 1969). The goal of FORTRAN at that time was to enable the IBM 704 to accept a concise formulation of a problem in terms of a mathematical notation and to automatically produce a high-speed 704 executable program for the solution of the problem. The 704 FORTRAN system was issued early in 1957 by the programming research department of IBM (International Business Machines Corporation). As a measure of the improvements in compiler technology and programming language design we note these first compilers were estimated to have required 18 man-years to produce (Alfred 1986); today, a one student project, in a one semester graduate course in compiling, typically results in a better compiler and language. In June, 1958 a new version of FORTRAN with significant languages additions was released as FORTRAN II for the 704. The
following are the three most significant additions of FORTRAN II to FORTRAN I: (1) the subroutine concept exemplified by the SUBROUTINE, CALL and RETURN statements and the FUNCTION statement, (2) the COMMON statement and the END statement, and (3) the use of subprograms permitted the linkage to assembly coded programs.

FORTRAN systems for the IBM 709 and 650 were released late in 1958. In the same year another major scientific computing language, ALGOL 58, was developed. Among the more intriguing technical features of ALGOL 58 were (a) its relative simplicity; (b) the introduction of the concept of three levels of language including a reference language, a publication language and hardware representations; (c) the begin ... end delimiters for creating a compound statement; and (d) the flexibility of the procedure declaration as well as several other important ideas in programming language design.

Many versions of FORTRAN were developed in the early 1960s. In 1960, FORTRANs for the IBM 1620 and 7070 were released, and in 1962 FORTRAN IV was released on the IBM 7030. From 1960-1962, several computer manufacturers other than IBM began developing different versions or extensions of FORTRAN on their computers. By 1963, virtually all manufacturers had either delivered or committed themselves to producing some version of FORTRAN.

Because of the widespread use of FORTRAN, several things happened. The methods of implementation differed, not only between manufacturers but within the same manufacturer. The different techniques of implementation created difficulties for the users. FORTRAN standardization was needed. In May, 1962, the ASA (American Standards Association) X3.4.3 (= FORTRAN) Committee to develop an American Standard FORTRAN was formed. Two standards (known as FORTRAN
and Basic FORTRAN, which correspond roughly to FORTRAN IV and FORTRAN II, respectively) were approved in March, 1966 (Sammet 1969).

In the early 1960s, most of the computers were second generation built with singly packaged transistors on handmade circuit boards, had very small physical memory (4K to 256K) made of hand wired magnetic cores, and had punch card batch oriented operating systems. To run a job, a programmer would first write the program on paper, then punch it on cards, and then bring the card deck to one of the operators. In the earliest days, the FORTRAN compiler was off-line. If the FORTRAN was needed, the operator would have to get it and read it in from cards. It is interesting to note that the maximum number of continuation cards were 5 and 19 in ASA standard Basic FORTRAN and FORTRAN, respectively.

An international standard of FORTRAN was accepted for most practical purposes in October, 1965; but final approval was delayed by administrative problems and errors. It added another subset to the two existing levels. This became the FORTRAN known as FORTRAN 66. All later FORTRANs are upward compatible with it, and it kept most of the features of the earliest of the FORTRANs intact.

To solve the incompatibilities between FORTRAN II and FORTRAN IV, the SIFT (SHARE Internal FORTRAN Translator) was developed. SIFT was a program primarily written in FORTRAN which would provide the necessary changes in a FORTRAN II program to make it work correctly as a FORTRAN IV program. Because FORTRAN had been so popular and implemented on so many other machines, there is a need to translate other languages to FORTRAN. In particular, a number of ALGOL programs had been hand-translated to FORTRAN in order to test them when no ALGOL compiler was available.
At that time, the obvious advantages of FORTRAN were its practical
effectiveness for solving numerical scientific problems and its subsequent widespread
use with reasonable compatibility and conversion facilities. The biggest disadvantage
of FORTRAN was that it could not truly be extended in any clean way to provide the
additional facilities that were then desired (Sammet 1969). Thus, the development of
what is now called PL/I was started.

While FORTRAN became more and more popular, use of ALGOL also
grew rapidly. ALGOL 60 was approved at an International Conference held in Paris
in January, 1960. ALGOL made a large number of significant contributions to the
technology, although it was not as popular as FORTRAN. Among the more
important ones are: (a) block structure and defining the scope of variables, (b) formal
language definition, (c) recursive procedures, (d) a general simplicity combined with
power for stating computational processes, and (e) a requirement for the development
of better implementation techniques.

Although popular in Europe, and embraced by the embryonic computer science
community in the United States, ALGOL never became a major factor in most
scientific computing. It suffered from a lack of implementations, and those that were
available tended to be weak and badly supported. Even by the mid 1960s the amount
of existing FORTRAN code made it hard for scientific computing at the time to justify
using anything except FORTRAN.

The development of PL/I started in 1963. In order to remedy FORTRAN's
lack of character and alphanumeric data handling and provide for good interaction with
more modern equipment and operating systems, a joint Advanced Language
Development Committee of SHARE and IBM was formed in 1963. The original
purpose was to specify a "major advance in FORTRAN." In March, 1964, a
document entitled "Report of the SHARE ADVANCED LANGUAGE DEVELOPMENT COMMITTEE" was presented to SHARE (see Sammet 1969). A few modified versions were available later that year. In December, 1964 another drastically revised version of the language appeared and was named PL/I by IBM. In the middle 1960s, some modern technology became available. IBM introduced the system/360 which was the first major computer line to use (small-scale) integrated circuits. Core memory was still popular and expensive and all programs had to execute in core. The concept of virtual memory and fast swapping disks was proposed, but still to be developed. The 360 was a series of software-compatible machines ranging from the 1401 to the more powerful 7094. Furthermore, they were designed to handle both scientific and commercial computing. Thus, the 360 was an ideal machine to run PL/I programs. The first PL/I compiler for system/360 was issued in August 1966.

PL/I has made a number of significant contributions to the technology because it included virtually all the good features from ALGOL, COBOL, and FORTRAN. PL/I is the first language to address itself seriously to the problems arising from interacting with an operating system.

Some of PL/I's weaknesses were that it could only run on the largest of the IBM machines; it normally produced larger and slower executables than FORTRAN; and the language had too many features so that errors in intended input sometimes produced syntactically correct PL/I programs. In fact there were several inconsistencies in the PL/I specifications which went undetected for a decade.

In 1970, a structured programming language, PASCAL, was introduced as an experiment in programming language design by Niklaus Wirth. By the end of the 1970s PASCAL became the most widely taught programming language for computer science, but it gained little acceptance in science and industry. The first PASCAL
compiler became generally available in 1971. It was a one-pass compiler organized around a recursive-descent (top down) parser. Some of the syntax for PASCAL was designed to use this technique in order to ease implementation and enhance portability of the compiler. PASCAL's syntax specification was formally given by a grammar, a simple extension of which was used to construct the compiler; this helped insure correct implementations and a language with a consistent syntax. PASCAL was a much smaller language than FORTRAN, COBOL, ALGOL or PL/I. Keeping a language small has major advantages that were not widely perceived at the time. This project, directed by one person, was just the opposite direction than that taken in the PL/I development. In retrospect the PASCAL approach is far better.

Starting in the late 1960s many efforts were carried out to package numerical methods into program libraries. There were significant early efforts on ALGOL and FORTRAN. By the mid 1970s most efforts were all in FORTRAN. Only in the last five years has an interest in numerical computations in C started to develop. As an example of numerical subroutine library development, the linear system package – LINPACK was produced in the early 1970s, by a group of scientists from the Argonne National Laboratory, Argonne, IL, the University of New Mexico at Albuquerque, the University of California at San Diego, and the University of Maryland at College Park, and several other sites (Dongarra 1979). LINPACK is a collection of FORTRAN subroutines which analyzes and solves various systems of simultaneous linear algebraic equations. The subroutines were designed to be completely machine independent, fully portable, and to run at near optimum efficiency in most operating environments. There were many other major FORTRAN packages and libraries started at this time as well including: Eispack (Smith 1974), Minpack
(Moré 1980), Hompack (Waston 1987), IMAL and NAG among others. These are still widely used today.

The use of integrated circuits (ICs) and multiprogramming, spooling and time-sharing techniques marked 1965-1980 as the period for the third generation of computers. Because of improvements of IC technology, computer manufacturers made major improvements on price/performance of computers. The multiprogramming technique improved efficiency of the CPU. The spooling technique provided the ability to read jobs from cards onto the disk as soon as they were brought to the computer room. The technique for time-sharing reduced response time by providing each user an on-line terminal (Tranenbaum 1987).

Although the first serious time-sharing system (CTSS) was developed at Massachusetts Institute of Technology (MIT) in 1962, it did not really become popular until necessary protection hardware became widespread during the third generation. After success of the CTSS, Massachusetts Institute of Technology (MIT), Bell Labs, and General Electric developed MULTICS (MULTiplexed Information and Computing Service). Ken Thompson at Bell Labs developed a one-user version of MULTICS – UNIX on the PDP-7. UNIX was later moved to PDP-11. UNIX has been moved to more computers than any other operating system, and its use is still rapidly increasing (Tranenbaum 1987).

The general-purpose programming language C was originally designed for and implemented on the UNIX operating system, by Dennis Ritchie (Kernighan 1978). C is a relatively "low level" language. Its modern control flow and data structures make it very convenient to use, and very popular. Usually the C compiler is, in fact, written in C. The LR parsing (bottom up) technique is used in the implementation of most C compilers. LR parsers can recognize virtually all common programming
language constructs from simple context-free grammar specifications. They can detect a syntactic error as soon as it is possible to do so. Because of the widely available UNIX tools LEX and YACC, LR parsing is more widely used than recursive descent parsing for the construction of portable language transformation research tools.

Two important UNIX tools, LEX and YACC, for computer language support were developed in the mid 1970s. LEX (Lexical Analyzer Generator) was written by M. E. Lesk (Lesk 1975). YACC (Yet Another Compiler – Compiler) was developed by S. C. Johnson (Johnson 1978). LEX and YACC are somewhat dated but still good tools used to generate the scanner and parser of a compiler. Using LEX and YACC, one can expect to port a compiler for a new machine in a couple of months; this is a major contrast to the fact that the first compiler of FORTRAN took 18 staff-years to implement.

Through the years, FORTRAN applications have ranged from strictly numerically oriented applications to more general applications involving character and file manipulation. As a result, most computer manufacturers have extended their version of FORTRAN to include more advanced processing facilities. A new standard was then needed to replace FORTRAN 66. A draft of a new FORTRAN standard (known as FORTRAN 77) was distributed in March, 1976, and approved in 1977. The new FORTRAN standard did not absolutely replace older FORTRAN programs but increased the scope of the language in several areas including: (a) input/output facilities, (b) data declaration facilities, (c) subprogram facilities, and (d) the addition of an IF THEN ELSE control structure.

The first FORTRAN 77 compiler, f77, was released in 1978, after the specifications for FORTRAN 77 were approved. It was produced with YACC and written by a very small group of people. The compiler f77 used the C compiler.
back-end so that on UNIX, C and FORTRAN object modules can be linked into the same executable, which was not a usual feature on other systems at the time (and still is not always available). This configuration provided a portable FORTRAN which moved relatively easily anywhere UNIX went. The compiler is a bit dated today, but still functional on many UNIX systems. Twelve years useful life for a compiler is a world’s record!

In 1980, Niklaus Wirth released the Modula-2 programming language (Wirth 1980). Modula-2 was an evolution of PASCAL. It improved on the successes of PASCAL while adding a facility for expressing the relations between the major parts of programs. In addition, Modula-2 contains high-level access to low-level machine features for systems programming and coroutines for concurrent programming. It had good control and data structures. Modula-2 showed us what a modern procedural programming language can be. The grammar for Modula-2 is significantly smaller than that for PASCAL and many programming language design improvements are apparent.

Wirth never intended PASCAL to be used as a general purpose programming tool; it was more a research exercise in language design. PASCAL’s shortcomings prevented its use from expanding more into scientific programming. Modula-2 addresses most of these problems and is gaining in popularity, though it is not expected to overtake FORTRAN in scientific programming.

The LSI (Large Scale Integration) circuits came in the mid 1970s. It made the minicomputers (like PDP/11) possible and allowed greater memory size. The VLSI (Very Large Scale Integration) chips came in the early 1980s. A VLSI chip contains 100,000 to 10,000,000 transistors on a square centimeter of silicon.

The age of the personal computer dawned in the early 1980s. In terms of architecture, personal computers were not that different from minicomputers of the
PDP-11 class, but in terms of price they certainly were different. The networks of personal computers running network operating systems and distributed operating systems were developed in the mid-1980s. The programming languages such as Turbo-C and Turbo-PASCAL became popular on PC. PASCAL-SC, a scientific computing language, also became available on PC and IBM machines.

Starting in the early 1970s, vector super computers were introduced. They contained vector processors with vectorizing FORTRAN compilers from companies like Cray (Cray Computer Corporation, Colorado Spring, Colorado) and CDC (Control Data Corporation, South Minneapolis, Minnesota). In the 1980s, new computers were introduced rapidly. Most of them supported the UNIX system. Because the f77 compiler was a part of the UNIX system and rides along almost free when UNIX is ported to a new machine, both C and FORTRAN have been the most solid and radially available languages on new machines in the last decade. FORTRAN 77 was also produced by DEC (Digital Equipment Corporation, Maynard, Massachusetts) for the VAX computer family. Note there is no other language with the same level of compiler support or as many running programs and libraries.

DEC's efforts produced truly fine optimizing FORTRAN compilers and helped the VAX line dominate the engineering minicomputer market. DEC in the mid-1970s, until some management reversals in the early 1980s, showed a great openness with functional details of the operation of the VAXs and liberal discounts to universities, in particular University of California at Berkeley where the BSD series of UNIX software releases were developed and released at very low cost in source form. This resulted in virtually every university computer science department, and many engineering departments, acquiring VAXs and running UNIX. That environment providing
FORTRAN and C (and later a very weak PASCAL) is a significant historical factor in looking at today's environment.

FORTRAN was modified or extended in an attempt to modernize the language or improve the FORTRAN programming environment by several projects, for example RATFOR (Kernighan 1975), MORTRAN and TOOLPACK. Meanwhile, the battle for FORTRAN 8X was continuing.

The purpose of FORTRAN 8X is to promote portability, reliability, maintainability, and efficient execution of FORTRAN programs for use on a variety of computing systems (Secretariat 1989). The FORTRAN 77 is entirely contained within FORTRAN 8X. Among the additions to FORTRAN 77 in FORTRAN 8X, ten stand out as the major ones: (1) array operations; (2) arithmetic, logical and character operators extended to arrays; (3) intrinsic functions extended to arrays; (4) numerical control, such as portable control over precision specifications; (5) basic facilities for data abstraction; (6) basic module facilities; (7) new control structures, a CASE statement and improved loop control; (8) more freedom in FORTRAN program file format specifications; (9) pointers; and (10) the concept of language evolution.

The ASA (American Standards Association) X3.4.3 Committee started FORTRAN 8X soon after the FORTRAN 77 standard was finished and only finished last year! It was expected to take only a couple years because when FORTRAN 77 was approved it was already clear that more improvements were needed—Modula-2 and C and PASCAL were around for examples and matrix features to help vectorizing compilers were wanted. The long delay caused vendors, notable DEC, to add their own extensions. Even the f77 compiler was modified to accept VAX extensions. This is a problem today because several of the VAX extensions are incompatible with the FORTRAN standard.
We believe that FORTRAN will still be the most popular scientific programming language in the near future. "I don't know what scientists will be programming in the year 2000, but I know it will be called FORTRAN" has become a favorite saying.

The rapid developments in improving microcomputer VLSI technology that have occurred in the 1980s have helped keep FORTRAN in the main stream of scientific computing. During this time the chips have evolved from no hardware, floating point and about 10,000 Flops (Floating point operations per seconds) to math co-processor with 100,000 Flops, to built-in, floating point vector operations with a peak of 80,000,000 Flops. The latter performance is a rough match of the Cray-1, the fastest vector supercomputer in the late 1970s. The new super-chips cost a few hundred dollars, while the Cray-1 cost several million. Although the performance of the super computers has also risen in this time, their relative performance gains have not been nearly as dramatic. From a performance advantage over the microcomputers of a factor of 10,000 they have fallen to a relative advantage of 15. The costs (adjusting for inflation) of supercomputers have not fallen.

The natural question is, is a super chip based computer now enough? For more and more applications it will be. If it is not, can we tie 16 of them together and make a multicomputer with performance equal to the supercomputers, at a small fraction of the cost. Or, can we tie 1024 of them together and get the fastest computer ever built?

There have been over two dozen important, ever more powerful, microprocessor chips in the last decade. The time from the start of a design of a new computer to its manufacture has fallen from over five years to less than a year, because of microcomputer technology. Unfortunately, a computer needs more than hardware. It needs an operating system, programming languages and application programs. No
vendor can produce these things from the ground up at a rate fast enough to keep pace with the changes in hardware, and if you don't keep pace with the change in hardware you won't sell computers.

This is a case of UNIX being at the right place at the right time. It was the only operating system with the needed features and relative portability. In the mid-1980s a port of UNIX to a new machine, of reasonable architecture, by experienced people, took about six months for a small team of people. In one case a port was carried out in two months by four people. Much of the effort actually goes into porting the C compiler itself and adapting the device drivers. Almost every advanced architecture computer built in the last decade has had UNIX as its operating system. Thus by default the C and FORTRAN languages were there.

Starting in the 1990s we see several new moves in programming environments for multicomputers including STRAND (Foster 1990) and EXPRESS (ParaSoft 1988). These almost all take great care to incorporate the ability to use C and FORTRAN within them.

Based on our historical perspective, we conclude this chapter with a list of desirable features for a "modern" programming language. Of course, some items on this list are subjective, not everyone would include all of them, but we hope many would agree with most of the items.

1. The language should be small.
2. Tokens in the language should be easy to classify, without needing extended context.
3. The language should be free format.
4. Algorithms should be expressible as programs in a form as close to the notation that would be typically used in their abstract (mathematical) specifications as possible.

5. The language should allow concise specifications of algorithms.

6. Comments should be able to be placed freely in the program without hurting the structured appearance of the program.

7. Keywords should be reserved words.

8. Common control structures should be explicitly supported. Block structure for the control structures should be clear and easy to analyze.

9. Names should be allowed to be long, and include some characters like '_'.

10. The language should be case sensitive.

11. All variables and functions should be explicitly declared.

12. All declarations should appear at the top of the program and should not be intermingled with the executable statements.

13. Strong type checking should be available.

14. Parameter lists on function invocations should be type checked against the function definitions.

15. Independently compiled program modules with hidden data and functions should be supported.

16. Nested procedure definitions are better done by modules.

17. There should be both explicit and implicit control of dynamical objects. (Explicit as in alloc and free, implicit as in static vs. automatic) This implies that there should be pointers. Composite data structures (C like structures) should be supported.

18. The precedence of operators should be simple and help avoid common errors.
19. Input and Output, as well as mathematical functions (e.g. sin, cos) and system interfaces, should all be provided as part of standard external libraries, rather than as part of the language proper.

20. There should be a facility to provide the compiler with extra information about functions, which it can use during optimization.

21. There should be standard external definitions of machine / environment dependent values needed in application programs.

22. IEEE floating point arithmetic should be provided (Anonymous 1985).

23. Numerical exceptions should be able to be handled by the application program.

24. High level expressions on matrices (e.g. FORTRAN 8X) should be added to allow optimization for advanced computers.

25. A standard preprocessor should be available (e.g. the C preprocessor) for file inclusion, conditional inclusion and macro definitions.

26. As far as possible, the language should have a simple lexical structure, suitable for a LEX program, with as little forward context used as possible.

27. An LALR grammar should exist for the language, which should be clear and have a simple YACC extension suitable for use in a compiler. Simple ambiguities should be allowed, such as dangling else.

Note that most of this list fits Modula-2 quite well. Despite a few "unusual features" we think Modula-2 is nearly the ideal prototype of what a procedural programming language should be.

Our Core CTRAN is designed, in Chapter III, to be one of the simplest "modern" procedural scientific programming language along with guidelines suggested above.
CHAPTER III

INTRODUCTION TO CTRAN

This chapter is a summary of the various constituents from which CTRAN programs are constructed. Both Core CTRAN and the current FORTRAN Compatibility Options are presented. The end of the chapter mentions some of the future directions that could be taken. Appendix A contains a brief manual for Core CTRAN alone. It is assumed that the ANSI C preprocessor (Kernighan 1988) is available and CTRAN programs may use it freely.

3.1 Introduction

We have stated in Chapter II our point of view about what a "modern" computer language should be. The main purpose of this chapter is to introduce such a "modern" computer language - CTRAN. CTRAN is designed as a scientific language. It is intended to facilitate using existing FORTRAN subroutines within a C environment on multiprocessors. The principle CTRAN goals are to provide: (a) easy translation from CTRAN into C via simple LEX and YACC programs; (b) fairly direct translation from FORTRAN into CTRAN; (c) a high level of control structure support; (d) removal of many irregular features from FORTRAN; (e) support of scientific calculations not needing of input and output, leaving that to standard libraries; (f) type checking and argument count checking in subroutine and function calls; (g) a simple front end for parallel processing research; and (h) simple facilities for exploring extensions to FORTRAN.
For the consideration of FORTRAN compatibility, CTRAN contains some "bad" features, called FORTRAN Compatibility Options. These include arrays with ( ) and do loops without end_do, etc.

The first remark we make in final differences between CTRAN and FORTRAN is that CTRAN takes the C approach, all input and output is provided by function libraries, for the reason previously given.

Core CTRAN itself is a very small language, currently much smaller than C. It concentrates on providing a FORTRAN like environment for strictly computational application, with as many improvements in lexical and syntactical matters as possible, while still corresponding as closely as possible to well written FORTRAN. Of course the latter is very subjective.

An example of a CTRAN program is shown in Figure 3.1. Since IO is not part of CTRAN, we omit the IO statements here.

Converting FORTRAN to Core CTRAN by hand would still be very laborious, so FORTRAN Compatibility Options are introduced. We would not like to see most of these options used in CTRAN programming, but they appear often enough in reasonable FORTRAN programs, which we are interested in, to warrant attention.

The presentation below gives both the Core CTRAN and many FORTRAN Compatibility Options. Presenting them separately would highlight the compactness of CTRAN, but extend the presentation.

It is intended that many well written FORTRAN programs can be fed directly to a CTRAN to C translator, using the FORTRAN Compatibility Options and a good looking C program will be produced. All unidentified FORTRAN constructs should be
subroutine distance()
begin
integer bar[6], numb, thisb
real dist

do numb = 1, 6
   bar[numb] = 0
end_do
while (.true.) /* infinite loop */
   ... input an real value into dist ...
   if (dist .lt. 0.0) goto L300:
      numb = INT(dist)
   if(numb .gt. 5)
      ... output "dist is out of range" ...
   else
      bar[numb + 1] = bar[numb + 1] + 1
   end_if
end_while
do numb = 0, 5
   thisb = bar[numb + 1]
   ... output values of numb and thisb ...
end_do
L300: return
end_subroutine

Figure 3.1

able to be included and clearly marked in the output, for human intervention. More on this is in the Chapter on the CTRAN to C translator.

CTRAN and FORTRAN Compatible CTRAN are defined so that any FORTRAN syntax which is accepted as CTRAN should have exactly the same semantic meaning in FORTRAN and CTRAN.

3.2 CTRAN Language Characteristics

3.2.1 Character Set

The CTRAN character set consists of letters, digits and special characters. A letter assumes its conventional interpretation and is one of the following 52 characters:
A digit is also defined conventionally and is one of the following 10 characters:

0 1 2 3 4 5 6 7 8 9

When a numeric value is represented, digits are interpreted to the base 10. Letters and digits are referred to as alphanumeric characters. The CTRAN character set includes the special characters:

+ - * / \ ( ) [ ] , . ' _ : # =

and the nonprinting characters: space <SP>, tab <TAB> and new line <NL>. The names of characters are given in Table 1.

Other special characters may be used in CTRAN programs and as data. When used in CTRAN programs, special characters other than those defined above must be protected in character literals.

There are seven classes of tokens: names, constants, labels, newlines, keywords, operators, and separators. The blank and tab characters serve as token separators and to provide indentation (which is checked for minimal consistency). Elsewhere, between tokens, the blanks and tabs, as well as comments, may be used freely to improve readability. If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token in a legal input with single character lookahead.

3.2.2 Names

A symbolic name (or, simply name) is a series of characters, assigned by the programmer to refer to a programmer-defined entity, such as a variable, array or
Table 1
Names of Characters in CTRAN

<table>
<thead>
<tr>
<th>Character</th>
<th>Name of character</th>
</tr>
</thead>
<tbody>
<tr>
<td>(no graphic representation)</td>
<td>space</td>
</tr>
<tr>
<td>(no graphic representation)</td>
<td>tab</td>
</tr>
<tr>
<td>(no graphic representation)</td>
<td>new line</td>
</tr>
<tr>
<td>=</td>
<td>equals</td>
</tr>
<tr>
<td>+</td>
<td>plus</td>
</tr>
<tr>
<td>-</td>
<td>minus</td>
</tr>
<tr>
<td>*</td>
<td>asterisk</td>
</tr>
<tr>
<td>/</td>
<td>slash</td>
</tr>
<tr>
<td>\</td>
<td>back slash</td>
</tr>
<tr>
<td>(</td>
<td>left parenthesis</td>
</tr>
<tr>
<td>)</td>
<td>right parenthesis</td>
</tr>
<tr>
<td>[</td>
<td>left bracket</td>
</tr>
<tr>
<td>]</td>
<td>right bracket</td>
</tr>
<tr>
<td>,</td>
<td>comma</td>
</tr>
<tr>
<td>.</td>
<td>decimal point</td>
</tr>
<tr>
<td>:</td>
<td>colon</td>
</tr>
<tr>
<td>’</td>
<td>apostrophe</td>
</tr>
<tr>
<td>#</td>
<td>hash mark</td>
</tr>
<tr>
<td>_</td>
<td>underscore</td>
</tr>
</tbody>
</table>
procedure. A name is composed of a sequence of alphanumeric characters or underscores, the first of which must be a letter or underscore. The length of a name is unlimited. This is quite different from the names in FORTRAN or C. The maximum length of a name in FORTRAN is six. Note that only the first eight characters of an internal name are significant in K&R's C. The smallest mandated significance of names with external linkage is six characters. In ANSI C the minimum significance of all internal names is increased to 31 characters; the significance of external names remains the same.

The grammar for a symbolic name is:

\[
\text{name} : \text{letter letter_digits} \\
| \ ' ' \text{letter_digits} \\
| ; \\
\text{letter_digits} : /* empty */ \\
| \text{letter_digits} \text{letter} \\
| \text{letter_digits} \ ' ' \\
| \text{letter_digits} \text{digit} \\
| ;
\]

CTRAN names are case sensitive. A warning on the use of the same name with some alternation in case will be issued by the compiler.

3.2.3 Keywords

A keyword is a sequence of characters that is significant in CTRAN and has a special meaning. The CTRAN keywords are as follows:

begin do end_main function real
block double end_subroutine goto repeat
All CTRAN keywords are in lower case. Under the FORTRAN Compatibility Options a keyword in upper case or mixed case will be automatically converted to the corresponding lower case, and meanwhile a warning message will be issued. All CTRAN keywords are reserved. Any attempt of using keywords as user defined names will cause fatal compilation errors. There is also a list of names reserved for future use; these include all the FORTRAN and C keywords, such as alloc and dealloc, and a few others. Warnings on their use can be generated. This is intended to reduce problems with later extensions and confusions in mixing CTRAN, FORTRAN and C.

3.2.4 Statement Structure

Unlike FORTRAN, CTRAN program is a fairly free format language. There is a mild restriction on which column a statement can start to insure consistent indentation. There is no restriction on how many characters can be written in one line.

3.2.4.1 Comment Lines

A CTRAN comment line is determined as follows: (a) anything between the token #c and an <eol>; or (b) anything between the tokens /* and */. This
may span lines and include #c ... <eol> commented lines. The second type of comments do not nest. Comment tokens (#c, /* and */) have no effect in literals.

EXAMPLES:

1. #c  This is a subroutine only for a test.

2. /*

   *  This program includes one subroutine and two functions.

   */

   A comment may be placed anywhere in a program unit and does not affect the executable program in any way. An empty line or a line consisting of blanks and/or tabs is considered as a comment line.

3.2.4.2 Continuation Lines

   A continuation line contains a continuation of a CTRAN statement and can be defined as follows: (a) \ is the last token in the previous line; or (b) #> is the first token in the current line.

   The number of continuation lines of a CTRAN statement is unlimited. A token in a CTRAN program should not be separated by continuation lines. Otherwise, the compiler will recognize it as two tokens. There may be some case where the compiler still finds a valid program, just not the program intended.

3.2.5 CTRAN Statements

   All CTRAN statements must begin with a new line except when part of the logical if statement which incorporates certain CTRAN statements as part of its normal statement structure.
A CTRAN statement may have a statement label so that it can be referred to in another CTRAN statement. A statement label consists of a name followed by a colon. FORTRAN Compatible CTRAN also allows an unsigned integer followed by a colon. A label is placed either preceding the statement in the same line or in some control statements. The scope of a label is the program unit in which it is included and the same label may not be used to label more than one CTRAN statement. It is not necessary to label a CTRAN statement. However, only statements that have been labeled may be referred to in other CTRAN statements. A CTRAN statement may be labeled, even if it is not referred to and any or all statements may be labeled. Warnings can be generated for unreferenced labels on statements or unreferenced labels in control statements.

All CTRAN statements, except the assignments, begin with a keyword. The keyword is used to identify the type of statement. For example, the statement do 20: i = 1, p begins with the keyword do.

3.2.6 Program Organization

A program unit is either a main program or a subprogram. It determines the scope of variables and statement labels. Unless explicit measures are taken, an entity defined in one program unit is not known by the processor when processing another program unit. The main program is the program unit that receives control when an executable program is placed in execution by the processor. It has a main statement as its first statement and end_main as its last statement. An executable CTRAN program must contain exactly one main program unit. A subprogram is a program unit that receives execution control of the processor by being referenced or called. It begins with a function (or subroutine) statement and ends with an end_function (or
end subroutine) statement. A CTRAN program may contain zero or more subprograms.

A CTRAN program may call some external procedure that is compiled or interpreted independently of the calling program unit. An external procedure may exist as a non CTRAN subprogram. It may be written in FORTRAN, C, or some other language to be called by a CTRAN program unit. The specification is in development and will be based in the C calling convention, in a particular environment.

3.3 Data Types, Data Structures and Expressions

3.3.1 Data types

There are seven types of data permitted in CTRAN: (1) integer, (2) real, (3) double precision, (4) complex, (5) double complex, (6) logical, and (7) character.

Three kinds of entities of CTRAN have a data type: (1) constants, (2) variables, and (3) functions. A constant is a data value that does not change during the execution of CTRAN program. In CTRAN, the data type of a constant may be arithmetic, logical, or character. The data type of a constant is determined by the manner in which it is written. The constants in CTRAN are identical to those specified in the FORTRAN Standard (see Harry 1978), except that embedded spaces may not occur.

Five types of arithmetic constants are permitted: (1) integer, (2) real, (3) double precision, (4) complex, and (5) double complex. An arithmetic constant may be signed or unsigned.

A grammar for CTRAN arithmetic constants is as follows.

\[
\text{arithmetic constant} : \text{int_constant} \\
\quad | \text{real_constant}
\]
I  db_precision_constant
I  complex_constant
I  db_complex_constant
;
int_constant : sign unsigned_int_constant
I  unsigned_int_constant
;
real_constant : sign unsigned_real_constant
I  unsigned_real_constant
;
db_precision_constant : sign unsigned_db_precision_constant
I  unsigned_db_precision_constant
;
complex_constant : sign unsigned_complex_constant
I  unsigned_complex_constant
;
db_complex_constant : sign unsigned_db_complex_constant
I  unsigned_db_complex_constant
;
unsigned_int_constant : digits
;
digits : digit
I  digits digit
;
unsigned_real_constant : real_no_exp
  | real_exp
  ;

real_no_exp : ',' unsigned_int_constant
  | unsigned_int_constant '.'
  | unsigned_int_constant '..'
    unsigned_int_constant
  ;

real_exp : real_no_exp 'E' int_constant
  ;

unsigned_db_precision_constant : real_no_exp 'D' int_constant
  ;

unsigned_complex_constant : '(' real_constant ',' real_constant ')'
  ;

unsigned_db_complex_constant : '(' db_precision_constant ','
    db_precision_constant ')
  ;

The FORTRAN Compatibility Options allow the omission of the decimal point when there is an E or D in the constant. It also allows the use of lower case e or d in the constant. Thus, under FORTRAN Compatible CTRAN, 1E2, le2, and 3D-4 are all acceptable. We also have a FORTRAN Compatibility Option which accepts complex constants with integer types in the components such as (1., 2) and (1, 2.).
3.3.1.1 Integer Data

An integer data item is an exact representation of an integral number. In CTRAN, an entity with an integer data type may exist as a constant, a parameter, an integer variable, an element in an integer array, a value of a function reference, or the value of an expression. An integer constant may specify a positive, negative, or zero value and is written as an optional sign followed by a string of decimal numbers. Some valid and invalid cases of integer constant are shown in Table 2.

Table 2
Some Valid and Invalid Cases of Integer Constant

<table>
<thead>
<tr>
<th>Valid Integer Constant</th>
<th>Invalid Integer Constant</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>59.</td>
<td>contains a decimal point</td>
</tr>
<tr>
<td>90</td>
<td>3,193</td>
<td>contains imbedded comma</td>
</tr>
<tr>
<td>+1990</td>
<td>-3.141</td>
<td>contains a decimal point</td>
</tr>
<tr>
<td>-83190</td>
<td></td>
<td>and fraction</td>
</tr>
</tbody>
</table>

The maximum magnitude of an integer data item is determined by the size of the storage unit of the machine.

3.3.1.2 Real Data

A real data item is an approximation to a real number and is dependent on the machine representation of this type of data. In CTRAN, an entity with a real data type may exist as a constant, a parameter, a real variable, an element in a real array, a value
of a function reference, or the value of an expression. A real constant may specify a positive, negative, or zero value and is written as one of the following three forms: (1) a basic real constant; (2) a basic real constant followed by a real exponent; and (3) an integer constant followed by a real exponent.

A basic real constant is an optional sign followed by a string of decimal digits with a decimal point. A real exponent is the letter E followed by a signed or unsigned integer constant, denoting a base 10 multiplier. The value of the real constant is either the basic real constant, or in the case of the exponent, it is the basic real or integer constant multiplied by the power of 10 following the letter E in the exponent. Some valid and invalid cases of real constant are shown in Table 3 and Table 4 respectively.

<table>
<thead>
<tr>
<th>Table 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Some Valid Cases of Real Constant</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Valid Real Constant</th>
<th>Numeric Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.5</td>
<td>-1.5</td>
</tr>
<tr>
<td>3E5</td>
<td>+300000</td>
</tr>
<tr>
<td>-.123E-5</td>
<td>-.000123</td>
</tr>
<tr>
<td>5.</td>
<td>+5</td>
</tr>
<tr>
<td>-645.2E21</td>
<td>-645.2*10^{21}</td>
</tr>
<tr>
<td>1.23E0</td>
<td>+1.23</td>
</tr>
</tbody>
</table>

The maximum magnitude and precision of a real constant is determined by the size of the numeric storage unit and the method of representation.
Table 4

Some Invalid Cases of Real Constant

<table>
<thead>
<tr>
<th>Invalid Real Constant</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>missing a decimal point or exponent</td>
</tr>
<tr>
<td>1,23.4</td>
<td>contains a comma</td>
</tr>
<tr>
<td>.123-3</td>
<td>missing the exponent designator E</td>
</tr>
</tbody>
</table>

3.3.1.3 Double Precision Data

A double precision data item is also an approximation to a real number and is dependent on the processor's representation of data. The difference between real and double precision data is that with double precision data, more digits of precision are maintained than with real data in accordance with the IEEE floating point standard. In CTRAN, an entity with a double precision data type may exist as a constant, a parameter, a double precision array, a value of a function reference, or the value of an expression. A double constant may specify a positive, negative, or zero value and is written as one of the following forms: (a) a basic real constant followed by a double precision exponent; or (b) an integer constant followed by a double precision exponent.

A double precision exponent is the letter D followed by a signed or unsigned integer constant. The value of the double precision constant is defined similar to the value of a real constant. We now give some valid and invalid cases of double precision constants in Table 5 and Table 6 respectively.
Table 5
Some Valid Cases of Double Precision Constant

<table>
<thead>
<tr>
<th>Valid Double Precision Constants</th>
<th>Numeric Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D2</td>
<td>+100</td>
</tr>
<tr>
<td>.123D0</td>
<td>+.123</td>
</tr>
<tr>
<td>6.789D−2</td>
<td>+6.789*10−2</td>
</tr>
<tr>
<td>−4.D+10</td>
<td>−4 * 10^{10}</td>
</tr>
</tbody>
</table>

Table 6
Some Invalid Cases of Double Precision Constant

<table>
<thead>
<tr>
<th>Invalid Double Precision Constants</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>−1.23456789</td>
<td>missing D exponent designator</td>
</tr>
<tr>
<td>1.23456789E10</td>
<td>incorrect exponent designator</td>
</tr>
<tr>
<td>.123,456D7</td>
<td>embedded comma</td>
</tr>
<tr>
<td>9.87654321D</td>
<td>missing an integer constant following D</td>
</tr>
</tbody>
</table>

The maximum magnitude and precision of a double precision constant is determined by the sign of the numeric storage units, used for storage of the double precision value, and the method representation. CTRAN assumes compliance with the IEEE floating point standard.
3.3.1.4 Complex Data

A complex data item is an approximation to a complex number. It is dependent on the processor's representation of real data. In CTRAN, an entity with a complex data type may exist as a constant, a parameter, a complex variable, an element in a complex array, a value of a function reference, or the value of an expression. It occupies two consecutive numeric storage units in a storage sequence, representing a pair of values of real data type. Two components represent the real part and the imaginary part of the complex data item respectively.

A complex constant is a pair of either real or integer constants, separated by a comma, and enclosed in parentheses. (2E3, 4D5) is an invalid complex constant because that a double precision constant 4D5 is used. Some valid cases of complex constants are shown in Table 7.

Table 7
Some Valid Cases of Complex Constant

<table>
<thead>
<tr>
<th>Valid Complex Constants</th>
<th>Numeric Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3.14, -4.13)</td>
<td>3.14 -4.13i</td>
</tr>
<tr>
<td>(-10, 5)</td>
<td>-10 +5i (FORTRAN Compatibility Option)</td>
</tr>
<tr>
<td>(.4E1, -.89E-1)</td>
<td>4 -.089i</td>
</tr>
</tbody>
</table>
3.3.1.5 Double Complex Data

A double complex data item is also an approximation to a complex number. The difference between complex and double complex digits of precision are maintained than with complex data. A double complex constant is a pair of integers, real or double precision constants. Therefore, (2D3, 4D5) is a valid double complex constant of value 2000 + 400000i.

3.3.1.6 Logical Data

A logical data item may be used to denote a "true" or a "false" value. There are two logical constants .true. and .false. in CTRAN. They have the logical values true and false respectively.

The words true and false must be preceded and followed by periods. In FORTRAN Compatible CTRAN, mixed cases are allowed.

3.3.1.7 Character Data

A character data item in CTRAN is a string of zero or more characters. Each character in a character data item occupies one character storage unit, and successive characters in a character data item occupy correspondingly successive character storage units.

A character constant is a string of characters enclosed in apostrophe characters. The apostrophes, used as delimiters, are not part of character constant. Within a character constant, an apostrophe is represented by two successive apostrophes. The following are examples of valid and invalid character constants:
Table 8
Some Valid Cases of Character Constant

<table>
<thead>
<tr>
<th>Valid Character Constants</th>
<th>Length</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A good one'</td>
<td>10</td>
<td>a good one</td>
</tr>
<tr>
<td>'Don’t '</td>
<td>5</td>
<td>don’t</td>
</tr>
<tr>
<td>'July+ 1990'</td>
<td>10</td>
<td>July+ 1990</td>
</tr>
<tr>
<td>' '</td>
<td>0</td>
<td>&lt;empty&gt;</td>
</tr>
</tbody>
</table>

Table 9
Some Invalid Cases of Character Constant

<table>
<thead>
<tr>
<th>Invalid Character Constants</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>wrong</td>
<td>not enclosed in apostrophes</td>
</tr>
<tr>
<td>'close, but</td>
<td>terminating apostrophe missing</td>
</tr>
</tbody>
</table>

3.3.2 Data Structures

Currently, CTRAN has the same data structure as FORTRAN. They are variables, character substrings and arrays.
### 3.3.2.1 Variables

A variable is an entity with a variable name, a type and a value. In FORTRAN, a variable whose first character is I, J, K, L, M, or N implies an integer type. In C, by default, all variables have integer data type. It is quite different in CTRAN. In Core CTRAN undeclared variables are an error. FORTRAN Compatible CTRAN uses the same implicit type conversions as FORTRAN. Adding an implicit statement to FORTRAN Compatible CTRAN is straightforward and is being considered. The need for implicit typing would be easily avoided in an automatic FORTRAN to CTRAN translator, by adding explicit declarations for all undeclared variables. A declaration specifies a type, and is followed by a list of one or more variables of that type, as in:

- integer low, high, middle
- complex x, y

This declaration list could equally well be written as:

- integer low
- integer high
- integer middle
- complex x
- complex y

The latter form leaves more space for adding a comment to each declaration.

### 3.3.2.2 Character Substrings

Core CTRAN relies on a system library for C compatible string functions. FORTRAN Compatible CTRAN has a substring operator, like FORTRAN's. A character substring is a set of contiguous characters that exists as a part of a character
data item. A substring must not be empty and has a character data type. A substring is specified with a substring name by one of the following forms:

\[ v \ (e_1 : e_2) \]
\[ a \ (s [, s] ...) \ (e_1 : e_2) \]

where, \( v \) is a character variable name; \( a \) is a character array name; \( e_1 \) and \( e_2 \) are integer expressions; \( s \) is a subscript of array \( a \).

The expression \( e_1 \) denotes the leftmost character position of the substring relation to the beginning of the "parent" character data item while \( e_2 \) is the right most position. If the "parent" character data item has length \( len \), then the following must hold:

\[ 1 \leq e_1 \leq e_2 \leq len \]

For example, if the following is defined:

character A * 8, B (10, 5) * 8

\[ A = 'STRING A' \]
\[ B(2,4) = 'STRING B' \]

then the substring \( A(7,8) \) has a character value "A" and the substring \( B(2,4) \ (1 : 6) \) has a character value of "STRING".

3.3.2.3 Arrays

An array is a set of data items of the same type that occupies consecutive storage units. Each array has an array name, a data type, a set of data values termed array elements, the number of dimensions and a lower and upper bounds for each dimension.

In CTRAN, all arrays must be declared and dimensioned before use. The declaration of an array is made with a CTRAN statement of the following form:

\[ \text{type_specifier} \ \text{array_declarator} \]
The type_specifier is one of five CTRAN data types. The array_declarator consists of a symbolic name of the array followed by dimension specifications in a sequence of one or more dimension specifications separated by commas. A dimension specification is a pair of subscript bounds d1 and d2 separated by a colon, which must be valid integer constant expressions. If d1 is omitted, it is assumed to be one. The following are some examples of array declarations in CTRAN:

integer a[2, 4, 3], b[10]

double precision b[2:2*(1990+1)]

Note that FORTRAN uses () for both array and parameter delimiters. We think C handles this better. In C, [] is used for array delimiters, while () is used as parameter delimiters. Core CTRAN inherits C's use of [] and (). FORTRAN Compatible CTRAN additionally allows the FORTRAN convention. Like in FORTRAN, arrays of CTRAN are stored in column-major order.

3.3.3 Expressions

An expression is a sequence of operands, operators, and parentheses. There are four kinds of operators: (1) arithmetic, (2) character, (3) relational, and (4) logical. The CTRAN expressions are designed to be exactly the same as FORTRAN expressions, in terms of operators, procedure and types. Promotion of data types is a useful FORTRAN Compatibility Option, which follows the same pattern as FORTRAN's.

An arithmetic expression represents a numeric value obtained through a numeric computation. The simplest form of arithmetic expression is an arithmetic constant, an arithmetic parameter, an arithmetic variable reference, an arithmetic array reference, or an arithmetic function reference.
There are five binary arithmetic operators in CTRAN: **, /, *, - and +. There are unary arithmetic operators + and -. Their precedence and associativity is given later in Figure 3.2.

The operators with a higher priority are executed before operators with a lower priority. Arithmetic operands must specify values that have a data type of integer, real, double precision, complex, or double complex. The structure of arithmetic expression is given by the following grammar:

```
arithmetic_expr  :  arith_expression
   |  '(' arith_expression ')'
;

arith_expression  :  unsigned_arithmetic_constant
   |  variable_name
   |  array_element_name
   |  function_reference
   |  '+' arithmetic_expr
   |  arithmetic_expr
   |  '-' arithmetic_expr
   |  arithmetic_expr '+' arithmetic_expr
   |  arithmetic_expr '-' arithmetic_expr
   |  arithmetic_expr '*' arithmetic_expr
   |  arithmetic_expr '/' arithmetic_expr
   |  arithmetic_expr '**' arithmetic_expr
;
```

One of the FORTRAN Compatibility Options is automatic promotion of type, following the FORTRAN conventions. Options allow no warnings, warnings on
conversions within expressions, warnings on conversions across assignments and only warnings on promotion of characters are provided.

When operands of different types appear in expressions, they are converted to a common type according to the type promotion. A type promotion is defined as follows, from lower to higher type:

- character
- integer
- real
- double precision
- complex
- double complex.

If we encounter \( A \ op \ B \), where \( A \) and \( B \) do not have the same type, then the lower type is converted to the higher type before the operation is performed.

Core CTRAN is designed to be a strongly typed language, therefore explicit type conversions are needed. To do this, explicit conversion operators are provided. The general format of these operators is \( x2y( ) \), where \( x \) and \( y \) are \( a, i, r, d, c, dc \) (character, integer, real, double precision, complex, double complex). The parameter of this operator has type \( x \) and the return value has type \( y \).

The relational operators are:

- \( \text{.gt.} \), \( \text{.ge.} \), \( \text{.lt.} \), \( \text{.le.} \).

They all have the same precedence. Just below them in precedence are the equality operators:

- \( \text{.eq.} \), \( \text{.ne.} \).

which have the same precedence. Relational operators have lower precedence than arithmetic operators.
Denote by $A$ and $L$ arithmetical and logical types respectively. Let $X$ be either $A$ or $L$; Table 10 gives a summary of operators and type conversion.

**Table 10**

A summary of Operators and Type Conversion

<table>
<thead>
<tr>
<th>Operators</th>
<th>Input Types</th>
<th>Output Type</th>
<th>Association</th>
</tr>
</thead>
<tbody>
<tr>
<td>$()$ $[ ]$</td>
<td>$(X) \ X[I]$</td>
<td>$X$</td>
<td>right to left</td>
</tr>
<tr>
<td>**</td>
<td>$A1 \ ** \ A2$</td>
<td>$\text{max_type}(A1, A2)$</td>
<td>left to right</td>
</tr>
<tr>
<td>* /</td>
<td>$A1 \ op \ A2$</td>
<td>$\text{max_type}(A1, A2)$</td>
<td>left to right</td>
</tr>
<tr>
<td>+ – (unary)</td>
<td>$\text{op} \ A$</td>
<td>$A$</td>
<td>right to left</td>
</tr>
<tr>
<td>+ –</td>
<td>$A1 \ op \ A2$</td>
<td>$\text{max_type}(A1, A2)$</td>
<td>left to right</td>
</tr>
<tr>
<td>.gt. .ge. .lt. .le.</td>
<td>$A1 \ op \ A2$</td>
<td>$L$</td>
<td>left to right</td>
</tr>
<tr>
<td>.eq. .ne.</td>
<td>$A1 \ op \ A2$</td>
<td>$L$</td>
<td>left to right</td>
</tr>
<tr>
<td>.not. (unary)</td>
<td>$L$</td>
<td>$L$</td>
<td>right to left</td>
</tr>
<tr>
<td>.and.</td>
<td>$L1 \ op \ L2$</td>
<td>$L$</td>
<td>left to right</td>
</tr>
<tr>
<td>.or.</td>
<td>$L1 \ op \ L2$</td>
<td>$L$</td>
<td>left to right</td>
</tr>
<tr>
<td>.equiv. .nequiv.</td>
<td>$L1 \ op \ L2$</td>
<td>$L$</td>
<td>left to right</td>
</tr>
<tr>
<td>=</td>
<td>$X1 = X2$</td>
<td>$X1$</td>
<td>right to left</td>
</tr>
</tbody>
</table>

where $\text{max\_type}(A, B)$ is the higher type of $A$ and $B$.

Since CTRAN contains $=$ as an operator, an expression leftside $=$ rightside becomes a statement which we refer to as assignment statement. A statement with the form expression is called an expression statement. Adding the C operators op= to CTRAN would be simple.

3.4 Control Flow Statements

The control flow statements of a language specify the order in which computations are down. CTRAN provides a better set of control flow statements than FORTRAN does. CTRAN contains some C-like control flow statements such as *for*, *while* and *repeat-until*.
We will see that CTRAN control flow statements if, else, do, for, while, repeat-until may contain multiple statements which we refer to as a block. In order to maintain a good readability, Core CTRAN checks indentations of statements. For indentation, all statements in the same block at the same nesting level must have the same indentation, which is determined by the indentation of the first instruction in the block (which must be greater than or equal to the indentation of the next outer block). An optional warning will be issued for a statement with wrong indentation.

3.4.1 Goto statements

Core CTRAN contains an unconditional goto statement to direct program control to an executable statement indicated by the statement label specified. As in FORTRAN, transfer of control to the interior of a block from outside the block is prohibited. Transfers within a block and transfers from inside the block to outside the block may occur. There are two additional types of goto statements in FORTRAN Compatible CTRAN: the computed goto and the assigned goto. The same restrictions on transfers of control for the unconditional goto apply.

The syntax of the unconditional goto statement is:

\[
goto \ label
\]

where, label is the statement label of an executable statement appearing in the same program unit. When the unconditional goto statement is executed by the processor, program control is transferred to the specified statements and normal sequential execution continues from there.

The syntax of the computed goto statement is:

\[
goto (labellist) \ i \ \text{or} \ \text{goto} (labellist) , i
\]
where the *labellist* is a list of statement labels separated by commas and is an integer expression. When the computed goto statement is executed, program control is passed to the statement identifier by the $i$th statement label in the *labellist*. If $1 \leq i \leq n$, where $n$ is the number of labels in the *labellist*. If $i < 1$ or $i > n$, then program control continues with the next sequential statement following the computed goto statement.

Two valid goto statements are as follows:

```
    goto missouri:
    goto (s20:, s30:, s20:), index+1
```

Currently, transfers to nonexecutable statements are not allowed. Some FORTRAN Compatibility Options, reducing this restriction, will be added, if they are useful frequently enough.

### 3.4.2 Logical If Statements

The if statements are used to make decisions. The syntax of logical if statement is:

```
    if (expression) statement
```

The *expression* must be of type logical. If it evaluates to `.false.` then control is passed to the statement following the if statement. If the *expression* evaluates to `.true.` then the statement in the if is executed and then, assuming that does not transfer control elsewhere, then control is transferred to the statement following the if statement. The statement in the arithmetic if statement may not be one of the `if` or `looping` statements.

### 3.4.3 Block If Statements

The syntax of the block if statement is as follows:

```
```
if (expression)
    statements_1 /* this is an if block */
else
    statements_2 /* this is an else block */
end_if

where the else block can be omitted and the type of the expression must be logical or
alternately:

if (expression)
    statements /* this is an if block */
else_if (expression)
    statements /* this is an else_if block */
...
[ Additional else_if blocks ]
...
else
    statements
end_if

where again, the else block can be omitted and the type of the expression must be
logical.

The if-else statement is executed as follows:

1. The expression is evaluated.

2. If the value is .true., statements_1 are done. If it is .false. and if there is an
   else part, statements_2 are executed instead.

3. Program control is transferred to the end_if statement and normal sequential
   execution continues from there.
The CTRAN function in Figure 3.2 returns the maximum value between two parameters.

```fortran
real function max(a, b)
    real a, b
    begin
        if (a .gt. b)
            max = a
        else
            max = b
        end_if
        return
    end_function
```

Figure 3.2

Because the else part of an if-else is also optional, there is an ambiguity when an else is omitted from a nested if sequence. This is resolved in the usual way—the else is associated with the closest previous else-less if.

The if-else_if structure is the most general way of writing a multi-way decision. The expressions are evaluated in order; if any expression is true, the statements associated with it are executed, and this terminated the whole chain by passing program control to the end_if statement. The last else part handles the default case where none of the other conditions was satisfied. If no default action exists, then the else block can be omitted.

The example in Figure 3.3 illustrates a binary search program in CTRAN.

3.4.4 Arithmetic If Statements

FORTRAN Compatible CTRAN has an arithmetic if statement. The form of the arithmetic if statement is:

```fortran
if (expression) label1, label2, label3
```

integer function binsearch(x, n, v)
integer x, n, v[n]
begin
integer low, mid, high

low = 1
high = n
while (low .le. high)
    mid = (low+high)/2
    if (x .lt. v[mid])
        high = mid - 1
    else_if (x .gt. v[mid])
        low = mid + 1
    else
        binsearch = mid
        return
    end_if
end_while
return
end_function

Figure 3.3

where, \textit{expression} is an arithmetic expression of type integer, real or double precision, but not complex or double complex; and label1, label2 and label3 are statement labels. The same restrictions on control transfer as hold on the unconditional goto apply. The \textit{expression} is evaluated, programs control is transferred to statements labeled label1, label2 or label3 depending on whether the value of expression is less then, equal to or greater than zero, respectively. After program control is transferred to one of the specified statement labels, normal sequential execution continues from there.

3.4.5 Do Statements

\textsc{C} \textsc{T} \textsc{R} \textsc{A} \textsc{N} contains a \textsc{F} \textsc{O} \textsc{R} \textsc{T} \textsc{R} \textsc{A} \textsc{N} like do control flow statement. The syntax of do statement is:
do opt_label var = expr1, expr2 [, expr3 ]

statements

end_do opt_label

In CTRAN, we require var, an integer variable. The expressions expr1, expr2 and expr3 have data type in integer, real, or double. They represent the initial value, limit value, and the increment value for the do variable var, respectively. The increment expr3 is optional. It may be positive or negative but not zero. The default value of increment is one. The opt_label is a statement label, it is optional, but if a label appears on one of the do or end_do a matching label must appear on the other. Do loops can be nested but must not overlap. An example of do loop is given in Figure 3.4

```plaintext
do L1: i = 1, n
  if (s[i].lt. 0) goto L1:
  sum = sum + x[i]
  prod = prod * x[i]
end_do L1:
```

Figure 3.4

To execute a do statement, first the expressions expr1, expr2 and expr3 are evaluated and converted to integer type in the case they have different data type, and control parameters, m1, m2 and m3, respectively, are determined. The iteration count, ic, is initially:

$$ic = \max \left( \frac{-m2 - m1 + m3}{m3}, 0 \right)$$

Note that, ic = 0 if m1 > m2 and m3 > 0, or if m1 < m2 and m3 < 0. Before the execution of the loop begins the do variable is initialized to m1.
Loop processing begins by testing the interaction count. If it is zero, the do loop becomes inactive and execution continues with the first executable statement following the end_do statement. If iteration count is positive, the do loops are executed successively until the end_do statement. The iteration count is then decreased by one and the do variable is incremented by \( m_3 \). Execution continues with the loop processing procedure described above.

The end_do is considered part of the do construct. In Core CTRAN it may not have a label, it is expected that the do construct will be labeled and control will be passed to the next cycle of the loop or the loop terminated by use of a cycle_loop or exit_loop statement. It is illegal to alter the do variable within the loop.

FORTRAN Compatible CTRAN allows the end_do to have a label and control may be transferred to the end_do from anywhere within the loop using a goto. The effect of this transfer corresponds to a cycle_loop statement for the same do structure.

The FORTRAN DO structure suffers from many problems in understandability. This comes with the use of the same labeled statement to terminate several nested loops or using a goto to transfer control to a labeled DO loop terminator. The control transfer in these cases is not as apparent from the syntax as it could be. This is often the cause of logic errors which escape detection during program testing. Comments on the nature of FORTRAN DO loop behavior in unusual cases is a favorite topic of the USENET comp.lang.fortran newsgroup, where many experienced FORTRAN practitioners differ on what the FORTRAN behavior is in specific cases. These opinions are often supported by behavior of specific compilers, showing the problem runs deeper than a misunderstanding by a few application programmers.

Most recommendations on FORTRAN programming style encourage always ending a DO loop with a CONTINUE statement, in an effort to provide a structured
style corresponding to the CTRAN do. The FORTRAN DO allows a slightly more compact coding; however the risks do not appear worth the savings.

The new FORTRAN 8X standard introduces a more structured, optionally named, DO loop. The unusual introduction of an optional name followed by a ':' before the DO, serves to provide a means to reference the DO anywhere inside the loop in EXIT and CYCLE statements. The fact that the old syntax is kept as well makes this improvement in syntax a suspect for future problems.

The introduction of a second new type of DO in FORTRAN 77, the DO TIMES, seems reasonable in terms of eliminating an unneeded dummy variable; however, our preference would have been to choose a new keyword name.

The FORTRAN 77 DO is the only loop control structure provided in that standard and is very heavily used. Although the translation from the FORTRAN 8X DO structures to the Core CTRAN do is straightforward for an automated translator, hand translation would be tedious and error prone. FORTRAN Compatible CTRAN provides an option for accepting FORTRAN 77 style DO loops. This, the odd two goto statements and the arithmetic if are the worst features of FORTRAN Compatible CTRAN.

3.4.6 For Statements

CTRAN has the following C-like for statement:

\[
\text{for (expr1 ; expr2 ; expr3) opt\_label} \\
\text{statements} \\
\text{end\_for opt\_label}
\]

The opt\_label has the same usage as in do statement.
Since CTRAN allows '=' as an operator, expr1 and expr3 may be assignments. In fact, without a ',' operator between statements in CTRAN, most commonly, expr1 and expr2 are assignments or function expressions. The expression expr1 specifies initialization for the loop. The expression expr2 must be type logical and specifies a test, made before each iteration, such that the loop is exited when the expression becomes .false.. The third expression expr3 often specifies an incrementation which is performed after each iteration. Any of the three expressions can be omitted, although the semicolons must remain. If the test, expr2, is omitted, it is taken as permanently true, so:

```plaintext
for (; ; )
...
end_for
```

is a infinite loop. The subroutine in Figure 3.5 is a Shell sort for sorting an array of integers.

```plaintext
/* shell sort: sort v[1] . . . v[n] into increasing order */
subroutine shellsort (n, v)
integer n, v[n]
begin
integer gap, i, j, temp
for (gap = n/2; gap .gt. 0; gap = gap/2)
    for (i = gap; i .lt. n; i = i + 1)
        for (j = i - gap; (j .ge. 0) .and. (v[j] .gt. v[j+gap]); j = j - gap)
            temp = v[j]
            v[j] = v[j+gap]
            v[j+gap] = temp
    end_for
end_for
return
end_subroutine
```

Figure 3.5

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
3.4.7 While Statements

CTTRAN has another loop control flow statement, the while statement. It is very much like the while statement in C. The syntax is:

\[
\text{while (expr) opt_label}
\]

\[
\text{statements}
\]

\[
\text{end\_while opt\_label}
\]

The opt_label follows the same rules as the opt_label in the for statement. The expr must be logical. To execute a while statement, the expr is first evaluated. If it is true, statements are successively executed until the end\_while is reached and expr is re-evaluated. This loop continues until expr becomes false, at which point control is transferred to the first statement after the end\_while. The test takes place before each iteration. If expr is initially false, the while loop body is never executed. An example of while statement is given in Figure 3.6.

while (j .lt. 20) w10:
    swapj = temp(j) .gt. 0
    jumpj = temp(j) .lt. 0
    temp(j) = j
    if (jumpj) temp(j) = -j
    if (.not. swapj) exit_loop 70:
end_while w10:

Figure 3.6

3.4.8 Repeat Statements

The last loop in CTRAN, the repeat-until statement, is a post-test looping structure, evaluating the test condition at the bottom after making each pass through the loop body, so the body is always executed at least once. The repeat-until statement
looks like the same statement in PASCAL, which is equivalent to the do statement in C. The syntax is:

```c
repeat opt_label
    statements
until (expr) opt_label
```

The `opt_label` and `expr` are similar to those in the while statement. The statements are executed first, then `expr` is evaluated. If it is `true`, statements are executed again, and so on. If the expression is `false`, the loop terminates. The repeat-until is much less used than the while, for or do statements. But, testing the termination condition at the bottom makes repeat-until loop statements very special. It is from time to time convenient for use. We now give an example of repeat statement in Figure 3.7.

```c
repeat r1:
    swapj = temp(j) .gt. 0
    jumpj = temp(j) .lt. 0
    temp(j) = j
    if (jumpj)
        temp(j) = -j
    if (.not. swapj)
        exit_loop 70:
    until (j .lt. 20) r1:
```

**Figure 3.7**

### 3.4.9 Exit_loop Statements

It is often convenient to be able to exit from a loop. This is one of the valid uses of a GOTO in most languages. For this purpose, CTRAN provides a C-like break statement - `exit_loop`. The syntax of `exit_loop` statement is:

```c
exit_loop opt_label
```
where, opt_label is a optional statement label. If the opt_label is omitted, the exit_loop causes termination of the smallest enclosing while, for, do or repeat loop. The labeled exit_loop must occur within a labeled loop (with the same label) and terminates that loop. In either case, control passes to the statement following the terminated loop (also see Figure 3.7).

3.4.10 Blank Statements and Cycle-loop Statements

There is a conflict between C's continue and FORTRAN's CONTINUE. The C's continue causes the next iteration of the smallest enclosing loop to begin. On the other hand, FORTRAN's CONTINUE serves as a point of reference, and is frequently used in a DO loop to provide a terminal statement. Because of FORTRAN's heavy use of CONTINUE, CTRAN needs something like it to ease FORTRAN to CTRAN translation.

A blank statement is introduced for this purpose. The syntax is:

```
label:
```

Clearly, FORTRAN's continue statement can be translated to blank statement of CTRAN:

```
100 CONTINUE  -->  s100:
```

CTRAN also contains a C-like continue statement--cycle_loop. The syntax of the cycle_loop statement is:

```
cycle_loop  opt_label
```

The opt_label may be omitted. If it appears, it must be the label of an enclosing loop. If it does not appear it refers to the smallest enclosing loop. The usage of cycle_loop statement is illustrated in Figure 3.8.
while (i .lt. n) start:
    if (a[i] .lt. 0)
        i = i + 1
    cycle_loop start:
end_if
    sum = sum + a[i]
    i = i + 1
end_while start:

Figure 3.8

The cycle_loop statement causes the next interaction of the enclosing loop (for, while, do, repeat-until) to begin. In while and repeat-until, this means that the test part is executed immediately; in the for, control passes to the re-initialization step; in the do, control passes to the incrementation step of loop control procedures.

If more FORTRAN 8X is added to CTRAN there would be a FORTRAN Compatibility Option allowing EXIT and CYCLE to be used as well as exit_loop and cycle_loop.

3.4.11 Following The Loops

The end_do, end_while and until statements are all considered part of the loop block in CTRAN. In Core CTRAN they may not be labeled. In FORTRAN Compatible CTRAN they may be labeled. In that case they can only be referenced from inside the loop block and the effect of a GOTO transferring control there is the same as a cycle_loop (see 3.4.9) for that block. Like the if block structure, Core CTRAN does not allow transfer into the interior of any loop block from outside the block.

3.5 Program Structure

A CTRAN program includes the three types of elements: (1) one main program; (2) zero or more intrinsic functions; (3) zero or more external procedures.
3.5.1 Main Program

A main program is a program unit that receives control of the processor when an executable program is loaded for execution. There should be one and only one main program in an executable program. The initial statement of a main program is a main statement.

The syntax of main statement is:

\[ \text{main } \text{pname} \]

where pname is the symbolic name of the main program. A main program may be composed of any set of CTRAN statements except the external procedures. A main program is terminated by an \textit{end_main} statement.

3.5.2 Intrinsic Functions

A FORTRAN Compatible CTRAN Option provides many of the FORTRAN generic intrinsic functions. For example, abs( ) for absolute value, sin( ) for trigonometric sine, and float( ) for type conversion to real.

A reference to an intrinsic function takes the form:

\[ \text{fname } (a[, a]) \]

where, fname is the name of the intrinsic function and a is an actual argument. An actual argument may be any valid expression. If more than one argument is permitted, then all arguments must have the same type. The generic names are used to classify intrinsic functions that perform the same mathematical function. If a generic name is referenced, then the compiler substitutes a function call to a specific name, depending on the data type of argument(s).
There is an outline for a simple C++ like extension providing generic function support for user functions and subroutines. This extension also provides a means for the programmer to pass information for use in an interprocedural dependency analysis. This overcomes one of the main advantages of FORTRAN having intrinsic functions which are known to the compiler, rather than using a library unknown to the compiler as C does. Another extension that could be integrated in is the use of 'in line' or 'in function' code generation, modeled after features in EC (Katzenelson 1983). These options together would give an optimizing compiler a great deal of useful information to work with, with little extra effort on the part of a programmer. This all can be neatly tied together with a subset of Modula-2 like module definitions and declarations. Nothing in current CTRAN implementation would require serious change and fairly close correspondence with the FORTRAN 8X module definitions remain.

3.5.3 External Procedures

An external procedure is a program unit that exists as an independent entity; it may be coded in a non CTRAN language. We discuss only external procedures written in CTRAN.

3.5.3.1 Kinds of External Procedures

Two kinds of external procedures exist in CTRAN: function and subroutine. A function begins with a function statement, is followed by a function body and terminated by an end_function statement. It has the following form:
[ ftype ] function fname([a [, a] ..])
formal_argument_type_definitions
begin
local_variable_type_declarations
...

/* body of the function */
end_function

where the object fname is the external name of the function. The ftype is optional in
FORTRAN Compatible CTRAN and specifies the data type of the function name,
which determines the value referred to the calling program. It has one of the following
form: (a) integer, (b) real, (c) double, (d) complex, (e) double complex, (f) logical,
or (g) character. The a is a dummy argument that takes one of the following forms for
a function: (a) variable name; (b) array name; (c) dummy procedure name.

A subroutine has the following form:

subroutine sname [ ([a [, a] ..]) ]
formal_argument_type_definitions
begin
local_variable_type_declarations
...

/* body of the subroutine */
end_subroutine

where sname is the symbolic name of the subroutine, and a is a dummy argument that
is similar to that in a function statement.

3.5.3.2 Referencing an External Function

A reference to an external function takes the following form:

fname ([a [, a] ..])
where fname is the symbolic name of the external function and a is an actual argument. The actual argument a constitutes an argument list and must match order, number, and type with the corresponding dummy arguments in the referenced function. An actual argument in a function reference must be one of the following: (a) an expression, (b) an array name, (c) an intrinsic function name, (d) an external procedure name, and (e) a dummy procedure name.

3.5.3.3 Call Statements

A subroutine is referenced in a calling program unit with the call statement. The syntax of a call statement is:

\[
\text{call sname [ ([a , a] ... ] ) ]}
\]

where sname is the symbolic name of subroutine and a is an actual argument. The actual argument a should satisfy the same condition as in the reference of functions.

3.5.3.4 Return Statements

The return statement causes program control to be referred to the referencing program unit. It has the form:

\[
\text{return}
\]

Note that there is no alternate return in CTRAN. As in FORTRAN, the value returned in a function is that set by assigning a value to the name of the function. If the return value is not assigned before the return, it is undefined.
3.6 FORTRAN Compatibility Options

Many options can be added to CTRAN to enhance FORTRAN Compatibility, such as removing the restriction on not mixing definitions and declarations with a begin in functions and subroutines.

One important one is also accepting FORTRAN style keywords. In the case that they are made reserved words, instances like ELSE IF can be recognized by the lexical analyzer as else_if, with no more needed than the already known context, such as being in a literal.

Another required option is the conversion of FORTRAN comments and line continuations into CTRAN style. This option is rather dumb, it assumes anything which is a legal FORTRAN comment or continuation must be converted. Unlike the other options this has dangerous effects on a CTRAN program and should only be used on FORTRAN source.

The case for splitting of the CTRAN to C translator into two separate parts, a FORTRAN Compatible CTRAN to Core CTRAN translator and a Core CTRAN to C translator is mentioned in Chapter IV.
CHAPTER IV

A CTRAN-TO-C CONVERTER

4.1 Introduction

As we have seen in previous chapters, CTRAN is a language intended to facilitate using existing FORTRAN subroutines within a C environment on multiprocessors. Thus, Automatic conversion of CTRAN to C is desirable. This chapter describes ctran2c, a program that translates CTRAN to C. We have used ctran2c to convert various programs and subroutines to C automatically. Our ctran2c is at its early age. In order to make things simple, we consider only four basic CTRAN types: (1) integer, (2) real, (3) double, and (4) logical. More work will be done in the future.

The current ctran2c may be split into two stand alone parts. The first part is a FORTRAN Compatible CTRAN to Core CTRAN Translator, which will be close to a FORTRAN to Core CTRAN Translator for "well written" FORTRAN without input and output. The second part will be a translator from Core CTRAN to C. Though the two part translation process is slower this allows a very compact Core CTRAN to C translator which is better suited for experimentation. The most interesting question is how to maintain two grammars related in this manner, and keep them consistent, as extra features are added to both. A set of rules for maintaining both syntactic and semantic consistency is being looked into.
There are other tools for integrating FORTRAN in a C environment, such as f2c (Feldman 1990), and for producing a more structured FORTRAN, such as Ratfor (Kernighan 1975).

4.2 Interlanguage Conventions

4.2.1 Names

Note that there is no limit for the length of a CTRAN name, and names may contain underscores. Since C truncates names that are longer than 8 characters, ctran2c will issue a warning for those CTRAN names that have length greater than 8. Note that in ANSI C, names may have any length, and for internal names, at least the first 31 characters are significant; some implementations may make more characters significant. To avoid conflict with names of library routines and with names that ctran2c generates, CTRAN names may have one or two underscores appended. External names, that is, the names of CTRAN procedures, have a single underscore appended if they do not contain any underscore and have a pair of underscores appended if they do contain underscores. Thus CTRAN subroutines named abc, a_b_c, and abc_, result in C functions named abc_, a_b_c__, and abc___.

A simple addition to the system would be a name mapper which maps long CTRAN names to fixed length C names. This is no problem for internal names; for external names, this requires a means for determining when two different names hash to the same value. This problem can be handled naturally if module support is added.

Because our current goal is to translate FORTRAN and provide a tool for some optimization exercises, it was not felt that long names were critical to the first implementation.
4.2.2 Types

The C types used in CTRAN for all the types are shown in Table 11.

<table>
<thead>
<tr>
<th>CTRAN</th>
<th>Corresponding C types used in ctran2c</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical</td>
<td>int</td>
</tr>
<tr>
<td>character</td>
<td>char</td>
</tr>
<tr>
<td>integer</td>
<td>int</td>
</tr>
<tr>
<td>real</td>
<td>float</td>
</tr>
<tr>
<td>double precision</td>
<td>double</td>
</tr>
<tr>
<td>complex</td>
<td>struct{float r, i;} complex</td>
</tr>
<tr>
<td>double complex</td>
<td>struct{double r, i;} dcomplex</td>
</tr>
</tbody>
</table>

4.2.3 Return Values

A function of type integer, logical, real, or double must be declared as a C function that returns the corresponding type. A complex or double complex function is equivalent to a C routine with an additional initial argument that points to the place where the return value is to be stored. Thus,

```
complex function f(...)
```

is equivalent to

```
void f_(temp, ...)
complex *temp;
...
```

A character-valued function is equivalent to a C routine with two extra initial arguments: a data address and a length. Thus,
character *15 function g( . . . )

is equivalent to

\[
g_\text{result, length, . . . } \\
\text{char *result; } \\
\text{int length;} \\
\ldots
\]

and could be invoked in C by

\[
\text{char string}[15]; \\
\ldots \\
g_\text{string, 15, . . . }
\]

4.2.4 Arguments Lists

All CTRAN arguments are passed by address. In addition, for every non-function argument that is of type character, an argument giving the length of the value is passed. The order of arguments is: extra arguments for complex and character functions, an address for each datum or function and an integer for each character argument other than character-valued functions. Thus, the call in

```
character *5 string
integer a[3]
\ldots
call sample(b[2], string)
```

is equivalent to that in

```
char string[5];
int a[3];
\ldots
sample(&b[1], string, 5);
```

Note that the first element of a C array has subscript zero, but CTRAN arrays begin at one by default. Because CTRAN arrays are started in column-major order, whereas C arrays are stored in row-major order, ctran2c translates multi-dimensioned CTRAN arrays into one-dimensional C arrays and issues appropriate subscripting expressions.
The translation of arrays from CTRAN to C could be done using arrays in C, with the indices reversed to account for the change from column major to row major, but the use of equivalence and common make explicit subscript expressions in C safer.

4.3 FORTRAN Intrinsic Functions and IO

Beyond lexical conventions, two areas where FORTRAN and C greatly differ are in FORTRAN's generic intrinsic functions and FORTRAN's extensive built in input and output facilities.

C's approach is very minimalistic. Beyond the basic language all features are supplied by external libraries. With the advent of the ANSI C standard, many of these defector standard libraries are now formally standardized. FORTRAN is a much larger language, with most features needed to write complete applications included as part of its definition.

The advantage of the approach C takes is that the compiler implementation is much simpler, smaller and easier to port to new machines. By defining standard libraries, as is done in ANSI C, which can be almost entirely implemented in the language itself, any drawbacks in terms of portability are removed.

The advantage of the approach FORTRAN takes concerns execution efficiency. This occurs in several forms. For example:

1. The compiler can make direct use of inline code for special functions provided by hardware.

2. The compiler can use simpler calling sequences for intrinsic functions; most use only a few registers and are not recursive.

3. The compiler can use information on which intrinsic function arguments are not modified to produce a better dependency analysis for optimization or parallelization.
FORTRAN's intrinsic functions pose two type of problems for translation: (1) the resolution of generic functions, and (2) variable number of arguments in the max and min functions. The simplest solution to FORTRAN's generic functions, from the view of CTRAN, is to require all calls to generic routines to be classified in the FORTRAN to CTRAN translation process, and replaced by the appropriate non-generic function call. This is a little extra work for a FORTRAN to CTRAN automatic translator, but would not be desirable by hand. Another approach would be to build the FORTRAN intrinsics into CTRAN. This is fairly easy, but clutters the language and implementation. Rather than build the intrinsics into CTRAN, CTRAN will be extended to support overloaded function names, as is done in C++. At that time, the overload would be specified in module definitions and these would also include the ability to inform the compiler about information which would allow efficient code generation.

The FORTRAN 8X standard is in a final approval stage. The extensions needed to support FORTRAN 8X features as well as generic CTRAN functions are easy to provide if the target language becomes C++ instead of C. For the present CTRAN serves its purpose well and extensions will be driven by need as well as research interest.

In the translation, FORTRAN IO is simply commented out and replaced with C IO calls. For now, FORTRAN IOs must be hand converted to appropriate C IO calls.

4.4 Invocation Examples

To convert the CTRAN files main.ct and subs.ct, one might use the UNIX command:
% ctranc2c main.ct subs.ct

This results in translated files suffixed with .c, that is, the resulting C files are main.c and subs.c. To translate all the CTRAN files in the current directory, compile the resulting C, and create an executable program named myprog, one might use the following UNIX commands:

```
% ctranc2c *.ct
% cc -o myprog *.c $(CTRANLIB)/ctran.a -lm
```

Sometimes it is desirable to translate several CTRAN source files into a single C file. This can be easily done by using ctranc2c as a filter:

```
% cat *.ct | ctranc2c > onefile.c
```

4.5 Translation Examples

Ctran2c is based on an early version of a CTRAN compiler that we have written. The compiler produced a C parse-tree for CTRAN control flow statements. The parse tree is walked to output the C code after it is generated. Since there is no power operator ** in C, the power operator ** of CTRAN is translated into a CTRANLIB.a function call. The CTRAN expressions are directly translated into C without doing optimization. Therefore, some expression in the resulting C file may not be presented in the most efficient way.

We now give some examples that illustrate ctranc2c's translation. For the purpose of comparison, we start from FORTRAN version of each routine.

```fortran
FUNCTION DOT(N, X, Y)
INTEGER N
REAL X(N), Y(N)
DOT = 0
DO 10 I = 1, N
  10 DOT = DOT + X(I)*Y(I)
END
```

The corresponding FORTRAN Compatible CTRAN routine is:

```c
FUNCTION DOT(N, X, Y)
  INTEGER N
  REAL X(N), Y(N)
  DOT = 0
  DO 10 I = 1, N
    10 DOT = DOT + X(I)*Y(I)
END
```
real function dot(n, x, y)
integer n
real x[n], y[n]
begin
integer i
dot = 0
do 10: i = 1, n
    dot = dot + x[i]*y[i]
end_do 10:
end_function

Ctran2c turns the above routine into:

/* Translated by ctran2c */
#include "ctran2c.h"

double dot_(n, x, y)
int *n;
real *x, *y;
{
/* System generated locals */
    int i_l;
    real ret_val;

/* Parameter adjustments */
    static int i;
    --y;
    --x;

/* Function body */
    ret_val = 0.;
    i_l = *n;
    for(i = 1; i <= i-1; ++i) {
        /* L10: */
        ret_val = ret_val + x[i]*y[i];
    }
    return ret_val;
} /* end dot_ */

The translated C always starts with a "translated by ctran2c" comment and an #include of ctran2c.h. Ctran2c forces the variable and procedure names to lower-case and appends an underscore to external name dot (to avoid possible conflicts with library names). The parameter adjustments "--x" and "--y" account for the fact that a C array starts at index 0. Unused labels are retained in commands for orienteering purpose. Within a function, CTRAN references to the function name are turned into references.
to the local variable `ret_val`, which holds the value to be returned. The local variable `ret_val` is generated by the translator. If the same name, `ret_val`, is used by the user, then the system will generate a temporary name by appending `ret_val` with appropriate number of underscores. In general, all temporary names generated by the system must not conflict with user names.

Recall that CTRAN does not contain IO statements. If, by mistake, the user has put some FORTRAN-like IO statements in a CTRAN program, then our ctran2c will comment the IO statements out and issue an error mark. If the following CTRAN routine is fed into ctran2c:

```
C Matrix Sim
subroutine matrix(a, b, c, m, n)
integer m, n
integer a(m, n), b(m, n), c(m, n)
begin
C internal variable
integer i, j
do 100: i = 1, m
   do 200: j = 1, n
      c(i, j) = a(i, j) + b(i, j)
      print 300 c(i, j)
   end_do 200:
end_do 100:
end_subroutine
```

then the corresponding C routine translated by ctran2c would be

```c
/* Translated by ctran2c */
#include "ctran2c.h"

/* Matrix Sum */
void matrix(a, b, c, m, n)
int *a, *b, *c, *m, *n;
{
   /* System generated variables */
   int a_dim1, a_offset, b_dim1, b_offset,
       c_dim1, c_offset, i_1, i_2;
   /* Local variable */
   static int i, j;
   /* Parameter adjustments */
   a_dim1 = *m;
```
a_offset = a_diml + 1;
a -= a_offset;
b_diml = *m;
b_offset = b_diml + 1;
b -= b_offset;
c_diml = *m;
c_offset = c_diml + 1;
c -= c_offset;

/* Function body */
i_1 = *m;
for (i=1; i <= i_1; i++) {
    i_2 = *n;
    for (j=1; j <= i_2; j++) {
        c[i + j * c_diml] = a[i + j * a_diml] + b[i + j * b_diml];
        print 300: c(i, j) */
    }
    /* L100: */
}
/* L200: */
c[i + j * c_diml] = a[i + j * a_diml] + b[i + j * b_diml];
/* print 300: c(i, j) */
/* 300: format (1x, f10.4) */
/* Error: CTRAN IO is not available! */
} /* end_matrix_ */

Therefore, hand translation of such IO statements is needed. For example this IO statement can be translated to C as:

```c
printf(" %10.4f
", c[i+j*c_diml]);
```

A LINPACK example and its CTRAN version together the corresponding C program are given in Appendix D for the comparison.

4.6 Conclusions

To conclude this chapter, we give a brief comparison between ctran2c and f2c. The f2c is a good program to automatically translate FORTRAN programs to C (or ANSI C or C++) programs (see Feldman 1990). It is based on the f77 compiler. Quite a large amount of the f2c codes deals with FORTRAN IO statement translation. Since CTRAN does not have IO statements, ctran2c has a much smaller size. The current version of ctran2c is based on a simple YACC grammar. The CTRAN tokens are recognized by a very simple LEX program. The current ctran2c is actually an early
version of Core CTRAN to C translator. The FORTRAN Compatible CTRAN to Core CTRAN translator is even simpler because it is mainly a syntax translator. A FORTRAN Compatible CTRAN to Core CTRAN translator and a Core CTRAN to C translator can be implemented completely in a much smaller size than that of f2c.
Appendix A

A Core CTRAN Reference Manual
1. Introduction

This manual describes the Core CTRAN language on the UNIX environment. It is readily possible to write Core CTRAN programs which are portable to other environments. Core CTRAN programs are markedly easier to write, and to read, and thus easier to debug. Throughout this manual, by CTRAN we mean Core CTRAN.

2. Lexical conventions

There are seven classes of tokens: names, constants, labels, newlines, keywords, operators, and separators. Blanks, tabs, and comments as described below are ignored except as they serve to separate tokens.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token with a valid one character lookahead.

2.1 Comments

A comment in CTRAN is anything between a #c and an <eol>, or anything between the /* and */. Comments may have any sequence of characters. /* ... */ may span lines and include #c ... <eol> commented line but do not nest. A comment line may be placed anywhere in a program unit and does not affect the executable program in any way.

A blank line is a comment line.
2.2 Continuations

The continuation via \ as last token in previous line, or via the symbol #> as first token in the current line.

2.3 Names

A name is a sequence of letters and digits; the first character must be a letter. The underscore _ counts as a letter. The length of a name is not limited. Names are case sensitive; a warning on same name different cases will be issued.

2.4 Labels

Labels are names terminated by '. Labels without the' can not be the same as variable names. A label may appear at the beginning of a line or be used in any control statement.

2.5 Keywords

The following names are reserved for use as keywords, and may not be used otherwise (uppercase versions not accepted or allowed as identifiers):

<table>
<thead>
<tr>
<th>begin</th>
<th>do</th>
<th>end_if</th>
<th>for</th>
<th>precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>block</td>
<td>double</td>
<td>end_mail</td>
<td>function</td>
<td>real</td>
</tr>
<tr>
<td>call</td>
<td>else</td>
<td>end_subroutine</td>
<td>goto</td>
<td>repeat</td>
</tr>
<tr>
<td>character</td>
<td>else_if</td>
<td>end_while</td>
<td>if</td>
<td>return</td>
</tr>
<tr>
<td>common</td>
<td>end_do</td>
<td>entry</td>
<td>integer</td>
<td>subroutine</td>
</tr>
<tr>
<td>complex</td>
<td>end_for</td>
<td>equivalence</td>
<td>logical</td>
<td>until</td>
</tr>
<tr>
<td>cycle_loop</td>
<td>end_function</td>
<td>exit_loop</td>
<td>main</td>
<td>while</td>
</tr>
<tr>
<td>data</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The keywords block, common, data, equivalence and entry are not currently used but are reserved for future use.

3. Expressions

An expression is a sequence of operands, operators, and parentheses that specifies an arithmetic, a character, a relational, or a logical computation. The primary expressions are identifiers, constants, strings, or expressions in parentheses. The evaluation of an expression is always a value that is used in the execution of the CTRAN statement that contains the expression. The precedence of expression operators is the same as the order of the following subsections of this section, highest precedence first. The left or right associativity is specified in each subsection for the operators discussed there.

3.1 Power Operator

The power operator ** groups right-to-left. The usual arithmetic conversions are performed.

\[
\text{power_expression} \quad : \quad \text{expression} \quad \text{"**"} \quad \text{expression} \\
\]

The binary ** operator indicates exponentiation.

3.2 Multiplicative Operators

The multiplicative operators * and / group left-to-right. The usual arithmetic conversions are performed.
multiplicative_expression : expression '*' expression
| expression '/' expression
;

The binary * operator indicates multiplication. It is associative and expressions with several multiplications at the same level may be rearranged by the compiler.

The binary / operator indicates division. The result of an integer division operation is a value of type integer.

3.4 Unary Plus and Minus Operators

The operand of the unary + or − operator must have arithmetic type. The result of unary plus expression is the value of the operand, while the result of unary minus expression is the negative of its operand. The unary + and − group right-to-left.

unary_expression : '＋' expression
| '－' expression
;

3.5 Additive Operators

The additive operators + and − group left-to-right. The usual arithmetic conversions are performed.

additive_expression : expression '＋' expression
| expression '－' expression
;
The result of the + operator is the sum of the operands. It is associative and expressions with several additions at the same level may be rearranged by the compiler.

The result of the − operator is the difference of the operands.

3.6 Relational Operators

The relational operators group left-to-right.

\[
\text{relational\_expression} : \text{expression} .\text{lt. expression} \\
| \text{expression} .\text{gt. expression} \\
| \text{expression} .\text{le. expression} \\
| \text{expression} .\text{ge. expression}
\]

The operators .lt. indicates less than; .gt. indicates greater than; .le. indicates less than or equal to; and .ge. indicates greater than or equal to. In an arithmetic relational expression, \( e_1 \) relop \( e_2 \), \( e_1 \) and \( e_2 \) are arithmetic expressions of type integer, real, or double. In Core CTRAN, \( e_1 \) and \( e_2 \) must have same type.

3.7 Equality Operators

\[
\text{equality\_expression} : \text{expression} .\text{eq. expression} \\
| \text{expression} .\text{ne. expression}
\]

The operator .eq. indicates equal to and the .ne. indicates not equal to. These two operators are exactly analogous to the relational operators except for their lower precedence. The expressions must have same type of integer, real, double precision, complex, or double complex.
3.8 Logical NOT Operator

\[
\text{logical_not_operator} : \ .\text{not}. \ expression \\
; \\
\]

The unary operator \(.\text{not}\). groups right-to-left and indicates logical negation. The operand must have type logical.

3.9 Logical AND Operator

\[
\text{logical_and_expression} : expression \ .\text{and}. \ expression \\
; \\
\]

The \(.\text{and}\). operator groups left-to-right. Its operands must have type logical. It returns \(.\text{true}\). if both its operands are true, \(.\text{false}\). otherwise.

3.10 Logical OR Operator

\[
\text{logical_or_expression} : expression \ .\text{or}. \ expression \\
; \\
\]

The \(.\text{or}\). operator groups left-to-right. Its operands must have type logical. It returns \(.\text{true}\). if either one of its operands is true, and \(.\text{false}\). otherwise.

3.11 Logical EQV and NEQV Operators

\[
\text{logical_eqv_operator} : expression \ .\text{eqv}. \ expression \\
| \ expression \ .\text{neqv}. \ expression \\
; \\
\]

The operator \(.\text{eqv}\). indicates logical equivalence, and \(.\text{neqv}\). indicates logical nonequivalence. They both group left-to-right. Their operands must have type logical.
3.12 Assignment Operator

\[ assignment\_expression : lvalues = expression \]

The operator = groups right-to-left. It requires a value as its left operand, and the type of an assignment expression is that of its left operand. The value of the expression replaces that of the object referred to by the lvalue. If both operands have arithmetic type, the right operand is converted to the type of the left preparatory to the assignment.

The lvalue is an expression referring to an object. An obvious example of an lvalue expression is a name.

3.13 Intrinsic Functions and Type Conversion Functions

FORTRAN intrinsic functions are supported in Core CTRAN. Core CTRAN is designed to be a strongly typed language, therefore explicit type conversions are needed. Explicit conversion operators are provided. The general format of these operators is \( x \rightarrow y() \), where \( x \) and \( y \) are a, i, r, d, c, dc (character, integer, real, double precision, complex, double complex). The parameter of this operator has type \( x \) and the return value has type \( y \).

4. Declarations

Declarations are used to specify the interpretation which CTRAN gives to each name; they do not necessarily reserve storage associated with the name. Declarations have the form

\[ declaration : type\_specifiers \ declarator\_list \]

;
The declarators in the declarators_list contain the names being declared. The type_specifiers consist of a sequence of type specifiers.

5. Type specifiers

The type_specifiers are

\[
\text{type_specifiers} : \text{character} \\
| \text{integer} \\
| \text{real} \\
| \text{double precision} \\
| \text{complex} \\
| \text{double complex} \\
| \text{logical} \\
\]

At most one typeSpecifier may be given in a declaration. All variables must be explicitly declared and dimensioned.

6. Declarators

The declarator_list appearing in a declaration as a COMMA-separated sequence of declarators, each of which may have an initializer.

\[
declarator_list : \text{declarator} \\
| \text{declarator_list} \ ',', \text{declarator} \\
; \\
declarator : \text{name} \\
| \text{array_declarator} \\
; 
\]
array_declarator : name [' dim_list ']
    ;

dim_list : dimension
    | dim_list ',' dimension
    ;

dimension : ubound
    | expr ':' ubound
    ;

ubound : '*'
    | expr
    ;

The expr in the declarator_list must be a constant expression of integer type.

Further the lower bound \( dL \) in a dimension specification \( dL : d2 \) must less then or equal to the upper bound \( d2 \). Note that characters \[ \] are used for array subscription.

7. Arrays

An array is a set of data items of the same type. Each array has the following properties:

1. an array name;
2. a data type;
3. a set of data values termed array elements;
4. a number denoting the number of dimensions;
5. an extent of each dimension; and
6. a lower and upper array bounds for each dimension.
An array can be specified by array name alone when reference or definition is made to the array as a whole, or an element in the array may be specified by the array name followed by an appropriate subscript.

8. Statements

All statements must begin with a new line, except when part of the logical IF statement, which incorporates certain CTRAN statements as part of its normal statement structure.

Like FORTRAN's statement, a CTRAN statement uses the carry-return as the end of line mark. Unlike FORTRAN, CTRAN program is a fairly free format language. There is a mild restriction on which column a statement can start to insure consistent indentation. There is no restriction on how many characters can be written in one line. For indentation, all statements in the same block at the same nesting level must have the same indentation, which is determined by the indentation of the first instruction in the block (which must be greater than or equal to the indentation of the next outer block).

Except as indicated, statements are executed in sequence.

Note, in order to distinguish the parameters and declarators, there is a keyword, begin, that is used between parameters and declarators.

8.1 Conditional Statement

The if statements provide the conditional facility in CTRAN; there are five forms of the if statements:

```
if_statements : if '(' expr ')' statement
               | if '(' expr ')' eoln statements eoln end_if
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The expression `expr` must be of type logical. If it evaluates to `.false.` then control is passed to the statement following the if statement. If the `expr` evaluates to `.true.` then the statement in the if is executed and then, assuming that does not transfer control elsewhere, then control is transferred to the statement following the if statement. The statement in the arithmetic if statement may not be one of the if or looping statements.

8.2 Do Statement

The do statement has the form

```
do_statement : do_prefix eolin statements eolin end_do opt_label

; do_prefix : do opt_label name '=' expr1 ',' expr2

| do opt_label name '=' expr1 ',' expr2 ',' expr3

;
```

The label in the do statement is the number of the last statement in range of the do loop. The `expr1` is an integer, real, or double expression representing the initial value given to the do variable. The `expr2` is an integer, real, or double expression
representing the limit value for the do variable. The $expr3$ is an integer, real, or double expression representing the increment value for the do variable.

8.3 For Statement

The for statement has the form

```
for_statement : for '(' assign ';' expression ';' assign ')' opt_label eoln statements eoln end_for opt_label ;
assign : expr '=' expr ;
```

The first `assign` in the for statement specifies initialization for the loop; the second `assign` often specifies an incrementation which is performed after each iteration. The expression in the for statement must be of type logical, which specifies a test, made before each iteration, such that the loop is exited when the expression becomes `.false.`.

If the `opt_label` are used in the statement, they should be the same labels to match.

8.4 While Statement

The while statement has the form

```
while_statement : while '(' expr ')' opt_label eoln statements eoln end_while opt_label ;
```

The statements is executed repeatedly as long as the value of the expression remains `.true.`. The test takes place before each execution of the statements.
If the *opt_label* is used in the statement, they should be the same labels to match.

8.5 Repeat Statement

The repeat statement has the form

```
repeat_statement : repeat opt_label eoln statements eoln
                 until (' expr ') opt_label
```

The repeat_statement is actually the repeat-until statement.

The statements are executed repeatedly the same as the while_statement, the difference with the while loop is the test takes place at the end of the each execution of the statement.

If the *opt_label* is used in the statement, they should be the same labels to match.

8.6 Goto Statement

Control may be transferred unconditionally by means of the statement

```
goto_statement : goto label
```

Transfer of control to the interior of a block from outside the block is prohibited.

8.7 Exit_loop Statement

The exit_loop statement has the form
exit_loop_statement : exit_loop opt_label

; The exit_loop without a label causes termination of the smallest enclosing
while, for, do, or repeat loop. The labeled exit_loop must occur within a labeled loop
(with the same label) and terminates that loop. In either case, control passes to the
statement following the terminated loop.

8.8 Cycle_loop Statement

The cycle_loop statement has the form

cycle_loop_statement : cycle_loop opt_label

; and causes control to pass to the loop-condition portion of the smallest enclosing do,
for, while statement; that is, to the end of the loop.

8.9 Blank Statement

The blank statement has the form

label

where label is a valid statement label. This statement is introduced for a compatibility
with CONTINUE statement of FORTRAN.

8.10 Return Statement

The return statement has the form

return_statement : return

;
The value returned in a function is that set by assigning a value to the name of the function. If the return value is not assigned before the return, it is undefined.

9. Function Statements

The form of the function statement is

\[
[\text{type}] \text{ function fun_name} \left( [d [, d ...]] \right)
\]

\[\text{....}\]

end_function

The type option has one of the following forms:

- character
- integer
- real
- double precision
- complex
- double complex
- logical

and specifies the data type of the function name, which determines the value returned to the calling program. The object fun_name is the symbolic name of the external function and unless specified. It is an external name and in a given executable program, it must be unique. The dummy argument d may be a variable name, array name, or dummy procedure name.

10. Subroutine Statements

The form of the subroutine statement is:
subroutine \texttt{sub\_name} [ ([d[,d]...])] \\
\hspace{1cm}.... \\
end\_subroutine

the object \texttt{sub\_name} is the external name of the subroutine of which the subroutine statement is its first statement, and \textit{d} is a dummy argument that takes one of the following forms for a subroutine:

1. variable name;
2. array name; and
3. dummy procedure name.

11. Main Program

A main program is a program unit that is not an external procedure. There can be only one main program in an executable program and that main program is identified by the fact that it does not have a function, subroutine, or block data statement as its initial statement. A main program may have a MAIN statement as its initial statement. The form of a main statement is:

\begin{verbatim}
main\ p\_name \\
\hspace{1cm}.... \\
end\_main
\end{verbatim}

where \texttt{p\_name} is the symbolic name of the main program; it must be a unique external name and must not be the same as any internal name in the main program.

12. Preprocessor and FORTRAN Compatibility CTRAN

The CTRAN compiler contains an ANSI C preprocessor capable of macro subroutine, conditional compilation, and inclusion of named file. Lines beginning with
# communicate with this preprocessor. These lines have syntax independent of the rest of the language; they may appear anywhere and have effect which lasts (independent of scope) until the end of the source program file. Those comment lines begin with #c and continuation of lines marked by => should be handled by the preprocessor.

As we mentioned in the chapters, Core CTRAN can be extended to FORTRAN Compatible CTRAN so that many FORTRAN Compatibility Options are allowed. One of the FORTRAN Compatibility Options is automatic promotion of type, following the FORTRAN conventions. Options allow no warnings, warnings on conversions within expressions, warnings on conversions across assignments and only warnings on promotion of characters are provided.

When operands of different types appear in expressions, they are converted to a common type according to the type promotion. A type promotion is defined as follows, from lower to higher type,

- character
- integer
- real
- double precision
- complex
- double complex.

If we encounter A op B, where A and B do not have the same type, then the lower type is converted to the higher type before the operation is performed.

The compiler directives for FORTRAN Compatibility Options may be set by macro variables, like "#define FORTRAN_COMPAT 0", means reporting FORTRAN Compatible CTRAN features with warnings. If "#define FORTRAN_COMPAT 1" is set, then FORTRAN Compatible CTRAN features are accepted without warnings.
Appendix B

A YACC Grammar for Core CTRAN
In this appendix, we present a YACC grammar for Core CTRAN. This is based on our implementation of the ctran2c translator. The grammar presented here accepts a set of strings that contains CTRAN language as a proper subset. Most of the type checking is done by the actions that we have omitted here. Those actions insure that only correct CTRAN programs can pass through the parser.

```
%token BEGIN
%token CALL CHARACTER COMPLEX CYCLE_LOOP
%token DATA DBCOMPLEX DO DOUBLE
%token ELSE ELSE_IF END_DO END_FOR END_IF END_FUNCTION
    END_MAIN END_WHILE END_SUBROUTINE EXIT_LOOP
%token FALSE FOR FUNCTION
%token GOTO
%token IF INTEGER
%token LABEL LOGICAL
%token MAIN
%token NAME
%token REAL REPEAT RETURN
%token SUBROUTINE
%token TRUE
%token UNTIL
%token WHILE
%token BITCON OCTCON HEXCON INTCON REALCON DBPCON
%token EOLN
%left SEMICOLON COMMA
%nonassoc COLON
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
\%
right \text{EQUALS}
\%
left \text{EQV NEQV}
\%
left \text{OR}
\%
left \text{AND}
\%
right \text{NOT}
\%
left \text{EQ NE}
\%
left \text{GT GE LT LE}
\%
left \text{CONCAT}
\%
left \text{PLUS MINUS}
\%
right \text{UNARY}
\%
left \text{STAR SLASH}
\%
right \text{POWER}
\%nonassoc \text{ LPAR RPAR LB RB}
\%
start \text{ctran\_program}
\%
\%
ctran\_program : /* empty */
| \text{ctran\_statements} \text{elon ctran\_program}
| ;
ctran\_statements
| : \text{opt\_label entry}
| | \text{opt\_label declaration}
| | \text{opt\_label executable\_stat}
| | \text{opt\_label BEGIN}
| | \text{opt\_label END\_FUNCTION}
opt_label END_SUBROUTINE
END_MAIN
;
entry : MAIN proname
MAIN proname progarg_list
SUBROUTINE entryname parameter_list
FUNCTION entryname parameter_list
type FUNCTION entryname parameter_list
;
proname : /* empty */
entryname 
;
entryname : name 
;
name : NAME 
;
progarg_list : LPAR RPAR
LPAR prog_arguments RPAR
;
prog_arguments 
: prog_argument
LPAR prog_arguments COMMA prog_argument
;
prog_argument 
: name
<table>
<thead>
<tr>
<th>name EQUALS name</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
</tr>
</tbody>
</table>

**parameter_list**: /* empty */

<table>
<thead>
<tr>
<th>LPAR RPAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LPAR parameters RPAR</td>
</tr>
<tr>
<td>;</td>
</tr>
</tbody>
</table>

**parameters**: parameter

<table>
<thead>
<tr>
<th>parameters COMMA parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
</tr>
</tbody>
</table>

**parameter**: name

<table>
<thead>
<tr>
<th>STAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
</tr>
</tbody>
</table>

**type**: type_specifiers leng_specifiers

<table>
<thead>
<tr>
<th>;</th>
</tr>
</thead>
</table>

**type_specifiers**

<table>
<thead>
<tr>
<th>INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL</td>
</tr>
<tr>
<td>COMPLEX</td>
</tr>
<tr>
<td>DOUBLE</td>
</tr>
<tr>
<td>DBCOMPLEX</td>
</tr>
<tr>
<td>LOGICAL</td>
</tr>
<tr>
<td>CHARACTER</td>
</tr>
<tr>
<td>;</td>
</tr>
</tbody>
</table>

**leng_specifiers**

<table>
<thead>
<tr>
<th>/* empty */</th>
</tr>
</thead>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
STAR expr

STAR LPAR STAR RPAR
;

declaration : type_specifiers declarator_list

;

declarator_list : declarator

    | declarator_list COMMA declarator

;

declarator : name

    | array_declarator

;

array_declarator

    : name LB dimension_list RB

;

dimension_list : dimension

    | dimension_list COMMA dimension

;

dimension : ubound

    | expr COLON ubound

;

ubound : STAR

    | expr

;

lhs : name

    | name substring
substring : LPAR opt_expr COLON opt_expr RPAR

opt_expr : /* empty */

function_arglist : /* empty */

function_args : expr

array_subscriptlist : /* empty */

array_subscript : expr

expr : uexpr

| name LPAR function_arglist RPAR
| name LB array_subscriptlist RB
| name LPAR function_arglist RPAR substring
| substring : LPAR opt_expr COLON opt_expr RPAR
| opt_expr : /* empty */
| function_arglist : /* empty */
| function_args : expr
| array_subscriptlist : /* empty */
| array_subscript : expr
| expr : uexpr

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
<table>
<thead>
<tr>
<th>complex_const</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
</tr>
</tbody>
</table>

**ueexpr** : lhs

<table>
<thead>
<tr>
<th>simple_const</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr addop expr %prec PLUS</td>
</tr>
<tr>
<td>expr STAR expr</td>
</tr>
<tr>
<td>expr SLASH expr</td>
</tr>
<tr>
<td>expr POWER expr</td>
</tr>
<tr>
<td>addop expr %prec UNARY</td>
</tr>
<tr>
<td>expr relop expr %prec EQ</td>
</tr>
<tr>
<td>expr EQV expr</td>
</tr>
<tr>
<td>expr NEQV expr</td>
</tr>
<tr>
<td>expr OR expr</td>
</tr>
<tr>
<td>expr AND expr</td>
</tr>
<tr>
<td>NOT expr</td>
</tr>
<tr>
<td>expr CONCAT expr</td>
</tr>
<tr>
<td>;</td>
</tr>
</tbody>
</table>

**addop** : PLUS

| MINUS |
| ;     |

**relop** : EQ

| GT    |
| LT    |
| GE    |
| LE    |
simple_const : TRUE
| FALSE
| INTCON
| REALCON
| DBPCON
| bit_const
|
bit_const : BITCON
| OCTCON
| HEXCON
|
complex_const
| : LPAR uexpr COMMA uexpr RPAR
|
executable_stats
| : executable_stat eoln
| executable_stat eoln executable_stats
|
executable_stat
| : assignment_statement
| goto_statement
| logical_if_statement
| block_if_statement
<table>
<thead>
<tr>
<th>do_statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>for_statement</td>
</tr>
<tr>
<td>while_statement</td>
</tr>
<tr>
<td>repeat_statement</td>
</tr>
<tr>
<td>call_statement</td>
</tr>
<tr>
<td>blank_statement</td>
</tr>
<tr>
<td>exit_loop_statement</td>
</tr>
<tr>
<td>cycle_loop_statement</td>
</tr>
<tr>
<td>return_statement</td>
</tr>
<tr>
<td>;</td>
</tr>
</tbody>
</table>

assignment_statement

assign : assign

assign : lhs EQUALS expr

;

goto_statement

: GOTO label

| GOTO LPAR label_list RPAR COMMA expr |
|                                      |
| ;                                    |

label_list : label

| label_list COMMA label             |
|                                      |
| ;                                    |

label : LABEL

;

logical_if_statement
: if_prefix executable_stat
  if_prefix eoln executable_stat
;

block_if_statement
  : if_prefix eoln executable_stats END_IF
  | if_prefix eoln executable_stats ELSE eoln executable_stats
    END_IF
  | if_prefix eoln executable_stats else_if_block END_IF
;
if_prefix
  : IF LPAR expr RPAR
  ;

else_if_block
  : elseif_prefix executable_stats
    | elseif_prefix executable_stats else_if_block
    | elseif_prefix executable_stats ELSE eoln executable_stat
    ;
elseif_prefix
  : ELSE_IF LPAR expr RPAR eoln
  ;
do_statement
  : do_prefix executable_stats END_DO opt_label
  ;
do_prefix
  : DO opt_label name EQUALS expr COMMA expr eoln
    | DO opt_label name EQUALS expr COMMA expr COMMA expr
      eoln
    ;
for_statement
  : for_prefix executable_stats END_FOR opt_label
 FOR LPAR for_list RPAR opt_label eoln


 assign SEMICOLON expr SEMICOLON assign


 while_prefix executable_stats END_WHILE opt_label


 WHILE LPAR expr RPAR opt_label eoln


 repeat_prefix executable_stats until_statement


 REPEAT opt_label eoln


 UNTIL LPAR expr RPAR opt_label


 CALL name

 CALL name LPAR RPAR

 CALL name LPAR argument_list RPAR

 argument_list : expr

 argument_list COMMA expr
; blank_statement
    : label
    ;
exit_loop_statement
    : EXIT_LOOP opt_label
    ;
cycle_loop_statement
    : CYCLE_LOOP opt_label
    ;
return_statement
    : RETURN
    ;
opt_label  : /* empty */
              | LABEL
              ;
opt_comma : /* empty */
              | COMMA
              ;
eoln      : EOLN
            ;
Appendix C

A Lexical Analyzer for Core CTRAN
We now present a CTRAN Lexical Analyzer. It is written using LEX—a Lexical Analyzer Generator. By showing how simple a CTRAN scanner is, we see that CTRAN does have a simple lexical structure which is one of the desirable features of "modern" programming language. Some of the action routines are omitted here.

```c
#define token(x) x

%{
#include "y.tab.h"

extern int lineno, continuation, indentation;
static int screen();
%
}

comment1 [c] ( [^\n] I [\)] ) * [\n]
comment2 "/*" "*/" * ([^*/] I [^*] "/" I "*/" [\*/]) * "*" * "*/"
letter [a-zA-Z_]
digit [0-9]
letter_digit [a-zA-Z_0-9]
label {letter_digit}+[\:]
int_num {digit}+
signed_int [\+-][int_num]
real_num ((digit)\^\.{digit}+l (digit)\.\^.) ([E] signed_int)?
db_num ((digit)\^\.{digit}+l (digit)+\.\^.) ([D] signed_int)
id {letter} {letter_digit}*
blank [\ ]
other .
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
".and." { return token(AND); }
".eqv." { return token(EQV); }
".eq." { return token(EQ); }
".neqv." { return token(NEQV); }
".ne." { return token(NE); }
".gt." { return token(GT); }
".ge." { return token(GE); }
".lt." { return token(LT); }
".le." { return token(LE); }
".or." { return token(OR); }
".not." { return token(NOT); }
".true." { return token(TRUE); }
".false." { return token(FALSE); }
"= " { return token(EQUALS); }
"+ " { return token(PLUS); }
"- " { return token(MINUS); }
"**" { return token(POWER); }
"*/" { return token(CONCAT); }
"*" { return token(STAR); }
="/" { return token(SLASH); }
"(" { return token(LPAR); }
")" { return token(RPAR); }
"[" { return token(LB); }
"]" { return token(RB); }
":: -- token(COLON) -
"", -- token(COMMA) -
";" -- token(SEMICOLON) -

^(comment1) { ECHO; lineno++; }  
(comment2) { ECHO; /* also set lineno */ }  
(db_num) { return token(DBPCON); }  
(real_num) { return token(REALCON); }  
(int_num) { return token(INTCON); }  
[bB][01]+ { return token(BITCON); }  
[oO][0-7]+ { return token(OCTCON); }  
[xXzZ][0-9a-f]+ { return token(HEXCON); }  
"double precision" { return token(DOUBLE); }  
"double complex" { return token(DCOMPLEX); }  
(label) { return token(LABEL); }  
{id} { return screenQ; }  
\"\n" | "\n" [ \t]* "#>" { ECHO; continuation = 1; lineno++; }  
^(blank)+ { ECHO; int i;  
   for (i=0; i < yyleng; i++)  
      if ( yytext[i] == '\t' )  
         indentation = 8 * ( 1 + indentation / 8 );  
   else  
      indentation++;  
}  
(blank)+ { ECHO; }  
\n { printf( "\n" ); continuation = 0; lineno++;}  

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
return token(EOLN);
}

{other} { return token(yytext[0]); }

%%%

/***** reserved word screener ****/

#include <stdio.h>

static struct rwtable { /* reserved word table */
    char * rw_name;
    int rw_yylex;
} rwtable[] = {
    /* characterize sorted */
    "begin", token ( BEGIN ),
    "call", token ( CALL ),
    "character", token ( CHARACTER ),
    "complex", token ( COMPLEX ),
    "cycle_loop", token ( CYCLE_LOOP ),
    "do", token ( DO ),
    "else", token ( ELSE ),
    "end_do", token ( END_DO ),
    "end_for", token ( END_FOR ),
    "end_function", token ( END_FUNCTION ),
    "end_if", token ( END_IF ),
    "end_main", token ( END_MAIN ),
    "end_subroutine", token ( END_SUBROUTINE ),
};
"end_while",
"exit_loop",
"for",
"function",
"goto",
"if",
"integer",
"logical",
"main",
"real",
"repeat",
"return",
"subroutine",
"until",
"while",

};

static int screen()
{
    struct rwtable * low = rwtable,
                  * high = END (rwtable),
                  * mid;

    int c;

    while ( low <= high ) {
        mid = low + ( high - low ) / 2;
    
        if ( ( c = strcmp ( mid->rw_name, yytext ) ) == 0 )
            break;
    
    }

    return 0;
}
return mid->rw_yylex;
else if ( c < 0 )
    low = mid + 1;
else
    high = mid - 1;
}
return token(NAME);
Appendix D

A LINPACK Procedure With Corresponding CTRAN and C Versions
For the sake of comparison, we list a LINPACK procedure with corresponding CTRAN and C versions in this appendix. The LINPACK procedure was written by G. W. Stewart, University of Maryland, Argonne National Lab. Most of the comments in the original procedure have been omitted as follows:

```c
SUBROUTINE SQRDC (X, LDX, N, P, QRAUX, JPVT, WORK, JOB)
INTEGER LDX, N, P, JOB
INTEGER JPVT(1)
REAL X(LDX, 1), QRAUX(1), WORK(1)

C INTERNAL VARIABLES
C
INTEGER J, JP, M, LP1, LUP, MAXJ, PL, PU
REAL MAXNRM, SNRM2, TT
REAL SDOT, NRMXL, T
LOGICAL NEGJ, SWAPJ

PL = 1
PU = 0
IF (JOB .EQ. 0) GO TO 60
   DO 20 J = 1, P
      SWAPJ = JPVT(J) .GT. 0
      NEGJ = JPVT(J) .LT. 0
      JPVT(J) = J
      IF (NEGJ) JPVT(J) = - J
      IF (.NOT. SWAPJ) GO TO 10
   IF (J .NE. PL) CALL SSWAP(N, X(1,PL), 1, X(1,J), 1)
      JPVT(J) = JPVT(PL)
      JPVT(PL) = J
      PL = PL + 1
10 CONTINUE
20 CONTINUE
   PU = P
   DO 50 JJ = 1, P
      J = P - JJ + 1
      IF (JPVT(J) .GE. 0) GO TO 40
      JPVT(J) = - JPVT(J)
50 CONTINUE
   IF (J .EQ. PU) GO TO 30
      CALL SSWAP(N, X(1, PU), 1, X(1, J), 1)
      JP = JPVT(PU)
      JPVT(PU) = JPVT(J)
      JPVT(J) = JP
30 CONTINUE
   PU = PU - 1
```
COMPUTE THE NORMS OF THE FREE COLUMNS.

IF ( PU .LT. PL) GO TO 80
DO 70 J = PL, PU
   QRAUX(J) = SNRM2(N, X(1,J), 1)
   WORK(J) = QRAUX(J)
70 CONTINUE
80 CONTINUE

LUP = MIN0(N, P)
DO 200 M = 1, LUP
   IF ( M .LT. PL .OR. M .GE. PU) GO TO 120
      MAXNRM = 0.0E0
      MAXJ = M
   DO 100 J = M, PU
      IF ( QRAUX(J) .LE. MAXNRM) GO TO 90
         MAXNRM = QRAUX(J)
         MAXJ = J
90 CONTINUE
100 CONTINUE
120 CONTINUE

QRAUX(M) = 0.0E0
IF (M .EQ. N) GO TO 190
NRMXL = SNRM2(N-M+1, X(M,M), 1)
IF (NRMXL .EQ. 0.0E0) GO TO 180
IF ( X(M,M) .NE. 0.0E0) NRMAL = SIGN(NRMXL, X(M,M))
CALL SSCAL(N-M+1, 1.0E0/NRMXL, X(M,M), 1)
X(M,M) = 1.0E0 + X(M,M)
LP1 = M + 1
IF (P .LT. LP1) GO TO 170
DO 160 J = LP1, P
   T = -SDOT(N-M+1, T, X(M,M), 1, X(M,J), 1)/X(M,M)
   CALL SAXPY(N-M+1, T, X(M,M), 1, X(M,J), 1)
150 CONTINUE
160 CONTINUE
170 CONTINUE

QRAUX(J) = QRAUX(J)*SQRT(T)
GO TO 140
CONTINUE
QRAUX(J) = SNRM2(N-M, X(M+1, J), 1)
WORK(J) = QRAUX(J)
CONTINUE
CONTINUE
CONTINUE
CONTINUE
C
SAVE THE TRANSFORMATION.
C
QRAUX(M) = X(M,M)
X(M,M) = -NRMXL
CONTINUE
CONTINUE
CONTINUE
RETURN
END
The corresponding CTRAN version is given below.

```c
subroutine sqrdc(x, ldx, n, p, qraux, jpvt, work, job)
integer ldx, n, p, job
integer jpvt[1]
real x[ldx, 1], qraux[1], work[1]
/* C */
/* C INTERNAL VARIABLES */
/* C */
begin
integer j, jp, m, Ipl, lup, maxj, pi, pu
real maxnrm, snrm
2, tt
real sdot, nrmxl, t
logical negj, swapj
/* C */
/* C */
pl = 1
pu = 0
if (job .eq. 0) goto 60
do 20: j = 1, p
swapp = jpvt[j] .gt. 0
negj = jpvt[j] .lt. 0
jpvt[j] = j
if (negj) jpvt[j] = -j
if (.not. swapj) goto 10:
if (j .ne. pi) call sswap(n, x[1, pi], 1, x[1, j], 1)
jpvt[pl] = j
pi = pi + 1
10:
20: end_do 20:
pu = p
do 50: jj = l, p
j = p - jj + 1
if (jpvt[j] .ge. 0) goto 40:
jpvt[j] = - jpvt[j]
if (j .eq. pu) goto 30:
call sswap(n, x[1, pu], 1, x[1, j], 1)
jp = jpvt[pu]
jpvt[pu] = jpvt[j]
jpvt[j] = jp
30:
pu = pu - 1
40:
50: end_do 50:
60:
/* C */
/* C COMPUTE THE NORMS OF THE FREE COLUMNS */
/* C */
if (pu .lt. pl) goto 80:
do 70: j = pl, pu
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
qraux[j] = snrm2(n, x[1, j], 1)
work[j] = qraux[j]

70:   end_do 70:
80:    
lup = min0(n, p)
do 200: m = 1, lup
   if (m .lt. pl .or. m .ge. pu) goto 120:
   maxnrm = 0.0E0
   maxj = m
do 100: j = m, pu
   if (qraux[j] .le. maxnrm) goto 90:
      maxnrm = qraux[j]
      maxj = j
90:    end_do 100:
100:   if (maxj .eq. m) goto 110:
call sswap(n, x[1, m], 1, x[1, maxj], 1)
qraux[maxj] = qraux[m]
work[maxj] = work[m]
jp = jpvt[maxj]
jpvt[maxj] = jpvt[m]
jpvt[m] = jp
110:   
120:   
qraux[m] = 0.0E0
   if (m .eq. n) goto 190:
   nrmxl = snrm2(n - m - 1, x[m, m], 1)
   if (nrmxl .eq. 0.0E0) goto 180:
   if (x[m, m] .ne. 0.0E0) nrmxl = sign(nrmxl, x[m, m])
call sscal(n - m + 1, 1.0E0/nrmxl, x[m, m], 1)
x[m, m] = 1.0E0 + x[m, m]
pl1 = m + 1
   if (p .lt. lp1) goto 170:
do 160: j = lp1, p
      t = - sdot(n - m + 1, t, x[m, m], 1, x[m, j], 1)/x[m, m]
call saxpy(n - m + 1, t, x[m, m], 1, x[m, j], 1)
   if (j .lt. pl .or. j .gt. pu) goto 150:
tt = 1.0E0 - (abs(x[m, j])/qraux[j])**2
   tt = amax1(tt, 0.0E0)
   t = tt
   tt = 1.0E0 + 0.05E0*tt*(qraux[j]/work[j])**2
   if (tt .eq. 1.0E0) goto 130:
qraux[j] = qraux[j] *sqrt(t)
goto 140:
130:   
140:   
qraux[j] = snrm2(n - m, x[m + 1, j], 1)
work[j] = qraux[j]
140:   
150:   
160:   end_do 160:
/* C */
/* C SAVE THE TRANSFORMATION */
/* C */
qraux[m] = x[m, m]
x[m, m] = -nrmxl

180:
190:
200:   end_do 200:
       return
       end_subroutine
The corresponding C version is:

```c
static int c_1 = 1;

void sqrdc_(x, ldx, n, p, qraux, jpvt, work, job)
float *x;
int *ldx;
int *n, *p;
float *qraux;
int *jpvt;
float *work;
int *job;
{
    /* System generated locals */
    int x_diml, x_offset, i_1, i_2, i_3;
    float r_1, r_2;

    /* Builtin functions */
    double r_sign(), sqrt();

    /* Local variables */
    static int negj;
    static int maxj;
    extern double sdot_(), snrm2_();
    static int j, m;
    static float t;
    extern int sscal_();
    static int swapj;
    extern int sswap_();
    static float nrnxi;
    extern int saxpy_();
    static int jj, jp, pl, pu;
    static float tt, maxnrm;
    static int lpl, lup;

    /* parameter adjustments */
    x_diml = *ldx;
    x_offset = x_diml + 1;
    x -= x_offset;
    -- qraux;
    -- jpvt;
    -- work;

    /* Function Body */
    /* C */
    /* C INTERNAL VARIABLES */
    /* C */
    /* C */
    /* C */
    pl = 1;
}
```
pu = 0;
if (*job == 0) {
    goto L60;
}
i_1 = *p;
for (j = 1; j <= i_1; ++j) {
    swapj = jpv[j] > 0;
    negj = jpv[j] < 0;
    if (negj) {
        jpv[j] = -j;
    }
    if (! swapj) {
        goto L10;
    }
    if (j != pl) {
        sswap_(n, &x[pl * x_dim1 + 1], &c_1, &x[j * x_dim1 + 1], &c_1);
    }
    jpv[j] = jpv[pl];
    jpv[pl] = j;
    ++pl;
L10:
/* L20: */
    pu = *p;
i_1 = *p;
for (jj = 1; jj <= i_1; ++jj) {
    j = *p - jj + 1;
    if (jpv[j] >= 0) {
        goto L40;
    }
    jpv[j] = -jpv[j];
    if (j == pu) {
        goto L30;
    }
    sswap_(n, &x[pu * x_dim1 + 1], &c_1, &x[j * x_dim1 + 1], &c_1);
    jp = jpv[pu];
    jpv[pu] = jpv[j];
    jpv[j] = jp;
L30:
    -- pu;
L40:
/* L50: */
};
L60:
/* C */
/* C COMPUTE THE NORMS OF THE FREE COLUMNS */
/* C */
    if (pu < pl) {
        goto L80;
    }
i_1 = pu;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
for (j = pl; j <= i_1; ++j) {
    qraux[j] = snrm2_(n, &x[j * x_dim1 + 1], &c_1);
    work[j] = qraux[j];
/* L70: */
}

L80:
lup = min(*n, *p);
i_1 = lup;
for (m = 1; m <= i_1; ++m) {
    if (m < pl || m >= pu) {
        goto L120;
    }
    maxnrm = (float) 0.;
    maxj = m;
i_2 = pu;
    for (j = 1; j <= i_2; ++j) {
        if (qraux[j] <= maxnrm) {
            goto L90;
        }
        maxnrm = qraux[j];
        maxj = j;
    }
    if ( maxj == m) {
        goto L110;
    }
    ss swap _(n, &x[m * x_dim1 + 1], &c_1, &x[maxj * x_dim1 +1], &c_1);
    qraux[maxj] = qraux[m];
    work[maxj] = work[m];
    jp = jpvt[maxj];
    jpvt[maxj] = jpvt[m];
    jpvt[m] = jp;
L90:
/* L100: */
};
if ( maxj == m) {
    goto L110;
}

L110:
L120:
qraux[m] = (float) 0.;
if (m == *n) {
    goto L190;
}

i_2 = *n - m + 1;
nrmxl = snrm2_(&i_2, &x[m + m * x_dim1], &c_1);
if (nrmxl == (float) 0.) {
    goto L180;
}
if (x[m+m * x_dim1] != (float) 0.) {
    nrmxl = r_sign(&nrmxl, &x[m + m* x+dim1]);
}
i_2 = *n - m + 1;
r_1 = (float)1. / nrmxl;
sscal_(&i_2, &r_1, &x[m + m * x_dim1], &c_1);
x[m + m * x_dim1] += (float)1.;
lp1 = m + 1;
if (*p < lp1) {
    goto L170;
}

i_2 = *p;
for (j=lp1; j <= i_2; ++j) {
    i_3 = *n - m + 1;
    t = -(double)dsdot_(&i_3, &x[m + m * x_dim1], &c_1, &x[m + j * x_dim1], &c_1) / x[m + m * x_dim1];
    i_3 = *n - m + 1;
    saxpy_(&i_3, &t, &x[m + m * x_dim1], &c_1, &x[m + j * x_dim1], &c_1);
    if (j < pl | j > pu) {
        goto L150;
    }
    if (qraux[j] == (float)0.) {
        goto L150;
    }
    r_2 = (r_1 = x[m + j * x_dim1], dabs(r_1))/qraux[j];
    tt = (float)1. - r_2 * r_2;
    tt = dmax(tt, (float)0.);
    t = tt;
    r_1 = qraux[j] / work[j];
    tt = tt * (float)0.05 * (r_1 * r_1) + (float)1.;
    if (tt == (float)1.) {
        goto L130;
    }
    qraux[j] *= sqrt(t);
    goto L140;
}

L130:

L140:
L150:
/* L160: */

L170:
/* C */
/* C SAVE THE TRANSFORMATION */
/* C */
qraux[m] = x[m + m * x_dim1];
x[m + m * x_dim1] = -(double) nrmxl;

L180:
L190:
/* L200: */

}
BIBLIOGRAPHY


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


