NASA Contractor Report 4565

The Adam Language: Ada Extended With Support for Multiway Activities

Arthur Charlesworth
University of Richmond
Richmond, Virginia

Prepared for
Langley Research Center
under Grant NAG1-774

National Aeronautics and
Space Administration
Office of Management
Scientific and Technical
Information Program
1993

ABSTRACT: The Adam language is an extension of Ada that supports multiway activities, which are cooperative activities involving two or more processes. This support is provided by three new constructs: diva procedures, meet statements, and multiway accept statements. Diva procedures are recursive generic procedures having a particular restrictive syntax that facilitates translation for parallel computers. Meet statements and multiway accept statements provide two ways to express a multiway rendezvous, which is an n-way rendezvous generalizing Ada's 2-way rendezvous. While meet statements tend to have simpler rules than multiway accept statements, the latter approach is a more straightforward extension of Ada. The only nonnull statements permitted within meet statements and multiway accept statements are calls on instantiated diva procedures. A call on an instantiated diva procedure is also permitted outside a multiway rendezvous; thus sequential Adam programs using diva procedures can be written. Adam programs are translated into Ada programs appropriate for use on parallel computers.

KEYWORDS: parallel programming, divide and conquer, reduction, multiway rendezvous, barrier, Ada, hypercube, recursion, ring.

This report was prepared in support of NASA unsolicited grant NAG-1-774, The Design and Implementation of a Preprocessor for the Ada Language to Support the Use of the Multiway Accept. The NASA technical monitor is C. Michael Holloway of the NASA Langley Research Center, Hampton, Virginia.

Programming in support of this project was performed primarily by Greg Morrisett, Jeff Michel, and Scott Shauf. The author is grateful for the support of NASA and for the unusual skill and dedication of Greg, Jeff, and Scott. The author also thanks the Institute for Parallel Computation at the University of Virginia for the use of its iPSC/2[1] hypercube and for arranging his 1989-90 sabbatical year at the Institute. Portions of this work during that year were funded by Jet Propulsion Laboratory grant 95722.

The prototype Adam translator is available via anonymous ftp from subdirectory **adam** of **urvax.urich.edu** No warranty covers the Adam translator or its documentation. However, the author wishes to learn of any errors or limitations. Email address: `charlesworth@mathcs.urich.edu`

[1]iPSC is a trademark of Intel Corporation

Contents

Chapter 1

Overview

After a brief motivation and a general description of Adam, this chapter describes three kinds of applications for which Adam is more suitable than Ada. The chapter also gives simple examples of Adam programs to illustrate the new language concepts, describes the target computers supported by the prototype translator, and discusses the relationship between Adam and Ada.

1.1 Motivation for Adam

Managers within large companies specify projects at a high-level, more in terms of *what* should be accomplished than in terms of precisely *how* everyone involved should do all necessary actions to accomplish the project. Assistants then refine any necessary details. Imagine what it would be like to manage thousands of employees for a company whose policy requires managers — without assistance — to describe projects directly in terms of the interactions between all pairs of workers on the project. Absurd as it may seem, Ada programmers must abide by a similar policy, since parallel communication in Ada must be expressed at the level of the two-way rendezvous. Even in the common situation where each worker is asked to execute the same code — the Single Program Multiple Data programming model — deciding *who does what when* can swamp the programmer with low-level detail.

The Adam language permits programmers to express algorithms either at a high-level of machine-independent parallelism involving cooperative, multiway activities, or at an even higher level in which multiway activities are completely implicit. The Adam translator then produces a semantically equivalent Ada program, with the required activities automatically converted down to the much lower level of two-way rendezvous. The name "Adam" derives from Ada plus support for multiway activities.

Effective computing requires adequate support from both hardware and software. The hardware support for high-speed computing is expected to be increasingly in the form of heterogeneous computing systems[22], systems that provide access to a variety of supercomputers and to workstation clusters. The lack of adequate software support for parallel computers has been termed the "parallel software crisis"[23]. In the long-run, ideal system software for heterogeneous computing systems should be capable of automatically shipping a program or program module to the most appropriate resource in the system for processing. The Adam language is a step in this direction. As explained later in the chapter, the prototype Adam translator generates three different kinds of target code, and for each of three different kinds of parallel computers there exist Adam programs for which the target code would achieve some form of asymptotically optimal performance. Moreover, the characteristics of these Adam programs that make them appropriate for a particular kind of parallel computer are typically known prior to run-time, assuming the desired number of processes and the length of certain vectors is known prior to run time.

1.2 General Description of Adam

Except for trivial restrictions on identifiers, Adam is an extension of Ada [26]. Thus all applications supported in Ada are also supported in Adam. In addition, Adam supports data parallel applications more conveniently than Ada, especially applications involving the kind of cooperative work described in this chapter.

Two related kinds of cooperative multiway activities are supported by Adam: the diva call and the multiway rendezvous.

- The **diva** call is a call on a high-level generic procedure that has a particular restrictive syntax, employs no explicit parallelism, uses recursion in a significant way, and whose correctness can be proven using strong induction. The name "diva" derives from the divide-and-conquer mechanism used by such calls. A diva call is useful for such applications as assigning values to the components of vectors and summarizing information concerning one or more vectors having the same length. Diva calls can be placed in an Adam program anywhere a procedure call is permitted in Ada, after the instantiation of its generic procedure.

- The **multiway rendezvous** generalizes the Ada two-way rendezvous by specifying that two *or more* tasks are synchronized during the execution of a (possibly empty) block of code. The simplest way to express a multiway rendezvous in Adam is using a **meet statement** in the task body of an array of tasks. When all members of the array reach the meet statement, the block of code is executed, after which the members resume their independent execution. Local variables of the participating tasks can be accessed within the meet statement, so a meet statement can resemble a meeting of the participating tasks. While the meet statement is sufficient for writing a multiway rendezvous, the Adam language also provides a second way to express a multiway rendezvous, the **multiway accept**, which is a straightforward generalization of the Ada accept. The multiway accept historically preceded the meet statement and has supported programming experiments comparing the two alternate approaches.

The diva call and the multiway rendezvous are integrated within the Adam language: diva calls are the only non-null statements that can be nested within the two Adam constructs supporting the multiway rendezvous. The block of code within a multiway rendezvous describes work to be performed during the multiway rendezvous and, since the tasks participating in the multiway rendezvous are synchronized during this work, this work is performed — logically speaking — by one or more additional agents. Of course, an implementation is permitted to implement this work in any way consistent with the semantics. Thus, whenever feasible, this work is performed by the same processes executing the tasks participating in the multiway rendezvous, so the work is performed cooperatively.

The multiway rendezvous, introduced in 1983[5], is a generalization of the Ada rendezvous [26] and also of the rendezvous in several other well-known concurrent languages such as CSP [16], DP [4], and SR [2]. It has been shown that none of the parallel languages supporting the concept of a rendezvous also provides adequate support for the multiway rendezvous [6]. The multiway rendezvous can be viewed as an extension of a barrier[14] that supports cooperative multiway activities.

The Adam language demonstrates the level of support for data parallelism provided by just two constructs: the diva procedure and the meet statement. Guy Steele, Jr. [25] lists several criteria a language should satisfy to adequately support data parallel programming, including the ability to express the following: elementwise addition, conditional operators, broadcast, reduction, parallel prefix, and permutation. Using diva procedures and meet statements satisfies all these criteria, for vectors.

Of course, vectors support a large class of important applications. They can model successive values of sensors over a fixed time interval, where the resulting data can be scientific (such as the temperature of a valve in a spacecraft), social (such as population), or commercial (such as stock market statistics). Additional one-dimensional phenomena include computer programs, DNA, natural language texts, and manuscripts of music. For some applications, diva procedures can be used with more general data structures. For instance, a diva procedure can be written to copy one two-dimensional array into another, by treating a two-dimensional array as a vector of vectors. But when a diva procedure is used to operate on a two-dimensional array either the rows will be treated independently of each other or the columns will be treated independently of each other, so full two-dimensional information is not being considered.

Creating a useful parallel programming concept is a balancing act among four primary goals:

- providing a level of abstraction with simple rules,
- supporting portability across parallel computers,
- providing sufficient expressibility for useful applications, and
- restricting the programmer to expressing high-level algorithms that permit generation of efficient code.

Some programming concepts, such as GAMMA [3], provide more expressibility than diva procedures. Some special-purpose programming concepts, such as primitives that satisfy single items in Steele's list, might be easier to translate into efficient parallel code than diva calls. Diva calls form a compromise between these two extremes, while providing a portable level of abstraction having simple rules.

Although expressed in an extension of Ada, the multiway activities represented by the diva procedure and the meet statement are presented as programming language concepts rather than just as constructs in a particular language. These concepts can be adapted for use in a wide variety of languages, just as the multiway branch concept — inspired by Hoare's case statement — has been widely adapted.

The important issue of how to provide the illusion of shared memory on a distributed memory architecture has been left to other researchers [21]. The Adam language design effort has focused on orthogonal issues.

The following sections describe applications conveniently supported in the Adam language. In addition to these applications, a diva call can be used to assign values to a vector, possibly using a computation based on corresponding values of vectors.

For simplicity, all complexity results in this report for calls on diva procedures assume that the computation within a single activation of the procedure takes constant time and that parameter passing for any parameter other than the vectors operated upon by a diva call (i.e., for any *nondynamic* parameter, as defined in Chapter 2) takes constant time. These conditions are met by all applications described.

1.3 Generalized Reductions

Languages supporting data parallel programming provide convenient ways to perform **reductions**; i.e., certain single computations across all components of a vector, such as finding the sum or product or maximum or minimum of the component values. In addition, several such languages also permit the programmer to use any function in a reduction, as long as the function satisfies certain properties. All such languages require the function to be associative and some also require the function to be commutative.

Any computation that can be performed using a reduction based on a programmer-defined function can be also be performed using a diva call. In addition, diva calls have the following advantages [8]:

- The correctness of a diva procedure can be proven in a straightforward way, using the same kind of strong induction argument naturally used in verifying a recursive procedure.

- Proving the associativity of a function can be very difficult, yet such a proof is essential for nontrivial programmer-defined functions. A separate proof of associativity is not required for diva procedures.

- Diva procedures are provably more general than reduction functions: there exists a diva procedure whose corresponding combining function is not associative.[1]

- The semantics of a diva procedure are natural and easy for the programmer to understand. The programmer has a simple sequential conceptual model for understanding these semantics: the model is based on the concepts of recursion and the nondeterministic divide. The latter delegates the choice of a division point in a vector used for recursive calls to the underlying implementation.

Independent of and complementary to the development of the nondeterministic divide are approaches that let the programmer specify an algorithm

[1]This follows from Example 2 of [9].

for selecting the division point in a vector [20]. The Adam language demonstrates what can be accomplished without specifying such an algorithm.

Diva calls can be used to perform a variety of computations on the sequences of values of vectors, such as:

- computing the number of peaks in a sequence, where a "peak" is a term that is greater than both its predecessor and successor,
- computing the position of the first term of a sequence of positive integers that is greater than a given value (or returning -1 if there is no such term),
- computing the maximum sum among the nonempty slices of a sequence of positive and negative integers, where a "slice" is a contiguous subsequence,
- computing the number of runs in a sequence, where a "run" is a slice whose values are increasing,
- computing the length of the longest plateau of a nondecreasing sequence, where a "plateau" is a slice of equal values,
- computing the length of the longest ascending slice of a sequence,
- computing the value of a term of one sequence corresponding to the maximum of another sequence,
- computing the length of the longest identical corresponding slice of two sequences,
- computing the length of the longest ascending slice of a sequence such that the corresponding terms of another sequence are also ascending,
- computing the largest increase among the pairs of adjacent terms of a sequence,
- computing the largest increase from one term of a sequence to a later term of the sequence, where the location of the first such term is an **in** parameter of the diva procedure,
- computing the largest increase from one term of a sequence to a later term of the sequence (and the two locations where such an increase first occurs), where the first such term varies over all terms of the sequence,

- computing the first record, in a sequence of records, whose fields have the most number of matches with the corresponding fields of a key,[2] and

- computing the length of the longest slice of one sequence whose terms are greater than the corresponding terms of another sequence (and the location where a slice of this length first occurs).

The final application can be used to compare the readings of two sensors over time, analogous to computing the longest winning streak of one sports team over another. Quicksort is an example of a divide-and-conquer strategy that does not have a straightforward treatment using diva procedures, since the division of a vector by quicksort occurs only after some initial processing of the vector.

1.4 Avoiding Certain Iterations

Consider the problem of computing the first location of a component of an unsorted vector of length n whose value differs the least from a given component's value, analogous to determining who in a group of n people has a height nearest a given person's height. This problem can be solved using a diva call, so it belongs to the class described in the preceding section. Now suppose the given component value is to vary over all the component values, analogous to asking everyone in the group of people to determine whose height is closest to their own. Such an application can be programmed by iteration, using a loop containing a diva call of the kind. described in the preceding section, so that a total of n diva calls are made.

But there is a simpler programming approach than such iteration, an approach that also provides guidance to the translator: place a diva call inside a meet statement and let the actual parameter corresponding to the given component value have a different value for different members of the array of tasks.[10] An Adam language rule assures us that such use of a diva procedure is just as easy to understand as a use of the same diva procedure in which different members of the array of tasks use the same actual parameter value, even though the net effect can be much more powerful. In addition, the target Ada produced by the Adam translator for a ring of n processors permits the n^2 computations to be performed in time proportional to n, which is optimal.

Many applications described in the preceding section can be parameterized this way. For example, for each term of a nondecreasing sequence one

[2]This application was suggested by Dana Richards of the U. of Virginia.

could compute the length of the longest plateau containing the given term. Here are additional examples:

- for each component in a vector, find the location of the first component to the right that has a greater value, if there is such a location.[3]
- sorting an array,
- performing a parallel prefix calculation,
- calculating the three accelerations on each of n astronomical bodies due to the masses and locations of the other bodies,
- letting each member of an array of n tasks send a different value to some other task in the array, and
- given a set of data, calculating which single piece of data, if omitted from the set, would cause the resulting variance to become the smallest.[4]

The first three of these additional examples, together with the example involving plateau lengths, are illustrated in Appendix A.

1.5 Loosely Synchronous Applications

When the term **multiway rendezvous** was introduced [5], its usefulness for a particular class of algorithms was illustrated. The algorithms in this class proceed by iterating stages, where part of the work of each stage is distributed among several processes, and where the only synchronization and communication is for sharing results among all these processes once during each stage (or, in general, a fixed number of times during each stage).

Loosely synchronous algorithms — a term used by Geoffrey Fox [13] — include the algorithms in this class. An example of such an algorithm is the straightforward parallel algorithm for solving the n body problem. The n body problem is that of finding the positions of n astronomical bodies at the end of a given time interval, given their initial positions, velocities, and masses, and assuming the only accelerations will be those due to mutual

[3]This problem was posed to us by David Nicol of the Institute for Computer Applications in Science and Engineering (ICASE) at NASA Langley Research Center, who had earlier needed to perform the calculation.

[4]This problem was posed to us by J. Van Bowen of the U. of Richmond, whose statistics Ph.D. dissertation includes a study of the variance of a set of data when one piece of data is omitted.

gravitational attraction. The straightforward parallel algorithm iterates over subintervals of the desired time interval and associates a process with each of the n bodies. Process i computes the new location of body i during each subinterval, using the locations and masses of all other bodies.

A few other examples of such algorithms, given in [5], include:

- The parallel farthest-insertion heuristic for the Euclidean traveling salesman problem [24].
- The parallel Floyd algorithm for finding the shortest distance between each pair of nodes in a network [11], given the length of each edge in the network.
- Parallel Jacobi iteration [19].
- The parallel Prim-Dijkstra algorithm for finding the minimum spanning tree of a network [12].

One approach to implementing such algorithms is to activate and deactivate processes before and after each stage. Another approach is to activate each process just once and to let the processes engage in a multiway rendezvous at each stage [6]. Advantages to the latter approach are:

- The number of process activations and terminations are greatly reduced; such operations can require significant amounts of processor time.
- Local variables are kept alive, resulting in less need for nonlocal variable accessing.
- With adequate support, such as that provided by the Adam language, the programmer can think at a higher conceptual level, leading to programs that are easier to write, read, debug, modify, test, and verify.

1.6 Examples of Adam Programs

This section gives simple examples of Adam programs to illustrate the key concepts, as well as a brief illustration of how these programs can be used.

As Nicklaus Wirth has observed:

> A successful language must grow out of clear ideas of design goals and of *simultaneous* attempts to define it in terms of abstract structures, and to implement it on a computer, or preferably even on several computers.[27]

The design of the Adam language reflects insights gained from simultaneously developing a translator for the language. The central purpose of the Adam translator has been to facilitate such insights. Thus the translator is basic research: a prototype rather than a production quality translator.

The first example given to illustrate a programming language traditionally involves one of the standard problems in computer science, such as computing a factorial, since to present a practical program initially would swamp the reader with details of a particular technical problem. Here is a VAX/VMS[5] terminal session that illustrates the use of the Adam translator in computing 5 factorial:

```
$ ADAM/MACHINE=LOOP_CUBE FACT_
$ ADA FACT_
$ ACS LINK FACT
%ACS-I-CL_LINKING, Invoking the VAX/VMS Linker
$ TYPE FACT.DAT1
5 5
5
$ ASSIGN FACT.DAT1 ADA$INPUT
$ RUN FACT
    120.0
```

The first command translates the file **FACT_.ADAM**, containing the Adam procedure **FACT**, into the equivalent Ada file **FACT_.ADA**, containing the Ada procedure **FACT**. The rest of the session uses the VMS Ada environment: **FACT_.ADA** is compiled and the result is linked, yielding **FACT.EXE**, which is run, producing the output.

The Adam translator can translate an Adam program into code for three different kinds of target machines: **IPSC2**, **LOOP_CUBE**, and **RING**. The user indicates the desired target machine by using the appropriate qualifier in the ADAM command. An explanation of these three target machines appears later in the chapter. The above example illustrates the form of data required for an Adam program. Preceding any data to be read by the Adam program must be two integers; these indicate the minimum and maximum number of processes desired in implementing a diva call. [For diva calls outside a multiway rendezvous, these data values are used to determine the number of nodes activated in the **IPSC2** implementation and the number of tasks activated in the **LOOP_CUBE** implementation.]

The program **FACT_.ADAM** can be written at least three ways: using a high-level diva call outside a multiway rendezvous and using either the meet statement or the multiway accept statement form of the multiway

[5]VAX and VMS are trademarks of Digital Equipment Corporation.

rendezvous. We now provide listings of these three programs. All three can be translated so the factorial is computed in parallel. The first program is the only one to use the third 5 in the sample data file, since it is the only one of the three Adam programs that explicitly reads any input. To illustrate how tasks participating in a multiway rendezvous can receive the common answer from a diva call, in the second and third versions of **FACT_.ADAM** presented here, these tasks print out their copy of the answer, resulting in five copies of `120.0` in the output. Altering the program to produce a single copy of the output is simple: the output statement would be embedded in an `if` statement that tests whether **MY_I** equals 1.

The following diva procedure is used in all three programs:

```
generic
  type RANGE_TYPE is range <>;
  type DYNAMIC_VECTOR is array(RANGE_TYPE) of FLOAT;
diva procedure FIND_PRODUCT (A: in DYNAMIC_VECTOR; PROD: out FLOAT) is
  -- Assign to PROD the * reduction of A
  INITIAL_PROD, FINAL_PROD: FLOAT;
begin -- FIND_PRODUCT
  if A'LENGTH = 1 then
    PROD := A(A'FIRST);
  else
    FIND_PRODUCT (A'INITIAL, INITIAL_PROD);
    FIND_PRODUCT (A'FINAL,   FINAL_PROD);
    PROD := INITIAL_PROD * FINAL_PROD;
  end if;
end FIND_PRODUCT;
```

The Adam extensions to Ada used in the first version of **FACT_.ADAM** are a generic diva procedure, a diva procedure instantiation, and a call on an instantiated diva procedure.

```
with FLOAT_TEXT_IO; use FLOAT_TEXT_IO;

procedure FACT is

  N: INTEGER;

  -- Place the declaration of diva procedure FIND_PRODUCT here


begin  -- FACT
  GET (N);
  declare
    subtype SUBRANGE is INTEGER range 1..N;
    type VECTOR is array(SUBRANGE) of FLOAT;
    diva procedure USE_PRODUCT is new FIND_PRODUCT (SUBRANGE, VECTOR);
    X: VECTOR;
    ANSWER: FLOAT;
  begin
    for I in SUBRANGE loop
      GET (X(I));
    end loop;
    USE_PRODUCT (X, ANSWER);
    PUT (ANSWER, 10, 1, 0);  NEW_LINE;
  end;
end FACT;
```

The Adam extensions to Ada used in the second version of FACT_.ADAM include a generic diva procedure, a diva procedure instantiation within the private part of a task type specification, and a task index declaration. The program also includes a meet statement containing a call on an instantiated diva procedure that uses hyphen notation to create a virtual vector using a local variable of the task type for the array of tasks.

```
with FLOAT_TEXT_IO; use FLOAT_TEXT_IO;
procedure FACT is
  PROD: FLOAT;
  N: constant INTEGER := 5;
  subtype SUBRANGE is INTEGER range 1..N;
  type VECTOR is array(SUBRANGE) of FLOAT;

  -- Place the declaration of diva procedure FIND_PRODUCT here.

  procedure PERFORM_COMPUTATION is

    task type WORKER_TYPE is
      private
        diva procedure USE_PRODUCT is new FIND_PRODUCT (SUBRANGE, VECTOR);
    end WORKER_TYPE;

    WORKER: array[MY_I: SUBRANGE] of WORKER_TYPE;

    task body WORKER_TYPE is
      MY_VALUE: FLOAT:= FLOAT(MY_I);
      MY_RESULT: FLOAT;
    begin -- WORKER_TYPE
      meet
        USE_PRODUCT (WORKER[-].MY_VALUE, MY_RESULT);
      end meet;
      PUT (MY_RESULT, 10, 1, 0);
    end WORKER_TYPE;
  begin -- PERFORM_COMPUTATION
    null;  -- Activate the array of workers.
  end PERFORM_COMPUTATION;

begin -- FACT
    PERFORM_COMPUTATION;
end FACT;
```

The Adam extensions to Ada used in the final version of FACT_.ADAM include a generic diva procedure, a multiway accept entry declaration, a diva procedure instantiation within the private part of a task specification, a task index declaration, and a multiway accept entry call. The program also includes a multiway accept statement containing a call on an instantiated diva procedure that uses hyphen notation to create a virtual vector using a formal parameter of the multiway accept.

```
with FLOAT_TEXT_IO; use FLOAT_TEXT_IO;
procedure FACT is
  N: constant INTEGER := 5;
  subtype SUBRANGE is INTEGER range 1..N;
  type VECTOR is array(SUBRANGE) of FLOAT;

  -- Place the declaration of diva procedure FIND_PRODUCT here.

  task PARENT is
    entry P[MY_I: SUBRANGE] (X: in FLOAT; RESULT: out FLOAT);
  private
    diva procedure USE_PRODUCT is new FIND_PRODUCT (SUBRANGE, VECTOR);
  end PARENT;

  task body PARENT is
    PROD: FLOAT;
    task type CHILD_TYPE;
    CHILD: array[MY_I: SUBRANGE] of CHILD_TYPE;
    task body CHILD_TYPE is
      MY_RESULT: FLOAT;
      MY_VALUE:  FLOAT := FLOAT(MY_I);
    begin -- CHILD_TYPE
      PARENT.P[MY_I] (MY_VALUE, MY_RESULT);
    end CHILD_TYPE;

  begin -- PARENT
    accept P[MY_I: SUBRANGE] (X: in FLOAT; RESULT: out FLOAT) do
      USE_PRODUCT (P[-].X, PROD);
    end P;
    PUT (PROD, 10, 1, 0);
  end PARENT;

begin -- FACT
  null;  -- Activate the PARENT task and thereby the array of CHILD tasks.
end FACT;
```

1.7 Available Target Codes

Throughout this section, a multiway rendezvous means a meet statement or multiway accept statement, p and n denote positive integers such that p is the number of processes used in implementing a diva call, and n is the length of the vectors used in a diva call. To ensure sensible use of processes, $n >= p$ must hold for each diva call; otherwise a run-time exception is raised. The value of p is determined by the data used with the Adam program, which begins with the minimum and maximum desired values for p. The programmer decides how the value of n is determined, whether by the data or by a constant or computation within the program.

An Adam program can be translated into one of three kinds of Ada code, targeted for three kinds of parallel computers, with the choice made by a compile-time directive. For input data satisfying the requirements given in the next paragraph, the choice of target computer does not affect the logical behavior of the program, it only affects performance, such as efficiency and speed of execution.

Here is a brief description of the three kinds of target Ada code:

- **The `IPSC2` target code for the Intel iPSC/2 hypercube, for which p must be a power of 2.** Adam programs most appropriate for the `IPSC2` are those in which a diva call only appears outside a multiway rendezvous. The combining work required for such calls is performed in log time on the iPSC/2, which is optimal (to within a constant factor). In the target Ada this combining work is performed by calls on the iPSC/2 `gopf` system function; thus the programmer has the benefits of both the high-level of abstraction provided by diva procedures and an efficient implementation in terms of the `gopf` call.

- **The `LOOP_CUBE` target code for any parallel computer having an Ada compiler, such that the computer permits hypercube interconnections between nodes, places different members of an array of tasks on different nodes when there are sufficiently many nodes, and simultaneously executes Ada rendezvous between pairs of such node programs whenever logically possible. There are no additional requirements on**

p and n. Adam programs most appropriate for the `LOOP_CUBE` are those for which each diva call within a meet statement involves the same nondynamic (defined at the beginning of the next chapter) in values and the Adam program contains a pragma asserting this fact. Given p processors, the combining work required for any diva call in such a program is performed in log time, which is optimal (to within a constant factor).

- **The `RING` target code for any parallel computer having an Ada compiler, such that the computer permits bidirectional ring interconnections between nodes, places different members of an array of tasks on different nodes when there are sufficiently many nodes, and simultaneously executes Ada rendezvous between pairs of such node programs whenever logically possible. The value of p must be even and $p = n$ must hold for any diva call within a meet statement.** Adam programs most appropriate for the `RING` are those in which a diva call occurs only within a meet statement and for which each diva call within a meet statement involves different nondynamic in values, since (given p processors) such calls will be executed in linear time, which is optimal efficiency (to within a constant factor).

The rest of this section provides additional details about these target parallel computers. The `IPSC2` target code consists of both a host and a node Ada procedure. The host Ada procedure obtains the largest subcube of the iPSC/2 within the size bounds specified by the data. If the minimum number of nodes desired is not available, the exception

```
ADAM_Z_MINIMUM_CUBE_SIZE_UNAVAILABLE
```

is raised. If $n < p$, the exception

```
ADAM_Z_DYNAMIC_VECTOR_LENGTH_SMALLER_THAN_MINIMUM_CUBE_SIZE
```

is raised. The user can then resubmit the target code (without changing it), and if the latter exception has been raised, the user will want to reconsider the appropriate level of parallelism specified in the data. For any diva call outside a multiway rendezvous, the host procedure ships the node procedure to each node of the subcube so the work of such a diva call is performed in parallel, with the result returning to the host. If $n = p$, the nodes combine their individual values. If $n > p$, the nodes independently perform some of the work of the diva call, working on (within one of) the same number of components of each vector used in the diva call, before combining their partial results. Since the iPSC/2 Ada compiler does not translate an Ada rendezvous into internode communication, any use of a multiway rendezvous is performed solely by the host program so there is no performance benefit for a diva call within a multiway rendezvous.

Although the `LOOP_CUBE` and `RING` target Ada codes can be compiled and executed on any computer having an Ada compiler, they are designed for parallel computers that should become available at some future date. Currently the author knows of no parallel computer having an Ada compiler that implements an Ada rendezvous as interprocessor communication so that large numbers of rendezvous can be executed simultaneously when this would be consistent with Ada semantics. Anticipating the ultimate availability of such a computer, the `LOOP_CUBE` and `RING` target codes incorporate restrictions on which members of a single-dimensional array `W` of tasks can call which other members, thereby simplifying the problem of mapping processes to processors, a problem an Ada system for such a computer would need to solve.

For the `LOOP_CUBE`, Adam code for a diva call and for a multiway rendezvous is translated into target Ada that only permits `W[`i`]` to rendezvous with `W[`j`]` if the binary codes for i and j differ in a single bit position, viewing the range of `W` as going from 0 to $p-1$. [But Adam places no arbitrary restriction on the programmer's choice of the range of `W`.] If $n > p$, a diva call outside a multiway rendezvous is translated into Ada that uses loops to perform independent work within each task prior to the intertask communication for the call, so that in the case when $p = 1$, such a diva call is executed as an Ada loop. The rest of this paragraph assumes the kind of parallel computer described in the preceding paragraph. For the applications described in Section 1.3, the target Ada performs the work of a diva call within a meet statement in time proportional to $n/p + log(p)$. For the applications described in Section 1.4, the target Ada performs the work of a diva call within a meet statement in time proportional to $n + plog(p)$; when $n = p$ these applications are better suited for translation to a ring.

For the `RING`, Adam code for a diva call within a meet statement is translated into target Ada that only permits `W[`i`]` to rendezvous with `W[`j`]` if i and j differ by 1 modulo p, viewing the range of `W` as going from 0 to $p-1$. Such a diva call is implemented so that, when the target code is executed on a suitable ring of processors, the number of rendezvous time steps required is $p/2$, where a rendezvous time step is the time required for executing a single Ada entry call (or several entry calls that can be executed simultaneously), and where the time for executing statements other than entry calls is ignored.[6] The target Ada performs the work of a diva call within a meet statement in time proportional to p ($= n$), on a suitable ring

[6] Scott Shauf, now at the U. of North Carolina at Chapel Hill, reduced the number of time steps from p to $p/2$.

of processors, as described above, for the applications described in **both** Section 1.3 and 1.4. For the latter class of applications, this is optimal, since p^2 computations are required for the given algorithms. For the applications in Section 1.3, the programmer should provide a pragma to ensure the target Ada achieves a smaller constant of proportionality; the pragma is explained in Section 3.2.

1.8 Relationship Between Adam and Ada

The Ada language has been criticized for being too elaborate [17]. Thus one should be very cautious in proposing an extension to Ada, especially if the extended features can interact with existing features of Ada to produce programs that are difficult to understand. The purpose of Adam is to make programs easier, rather than more difficult, to understand. Thus restrictions are placed on the types of interactions that can occur between the new and old features. For example, the multiway accept cannot be used to select options in an Ada select statement and calls on multiway accept entries cannot be used to select options in an Ada select statement. The Adam language, with these restrictions, fully supports the applications described in this report.

Adam is a superset of Ada and thus the syntax and semantics of all Ada reserved words, comments, statement forms, etc., are preserved by the Adam translator. The only restrictions on the pure Ada used by an Adam programmer are the addition of the reserved words **diva** and **meet** and the fact that a user-defined identifier cannot have the prefix `ADAM_Z` and cannot have length greater than 40. The translator uses `ADAM_Z` as a prefix for identifiers inserted into the translated Ada form of the Adam program. Since the Ada programs generated by the Adam translator have comments beginning with `--***` to explain the role of the Ada that has replaced extensions of Ada, the Adam programmer might wish to avoid using comments beginning with these characters so that the new comments are easier for a programmer to notice. Of course, there may be no need for a programmer to read an Ada program generated by the Adam translator.

Since the introduction of Adam in 1987, the language has supported the automatic assignment of indexes to an array of tasks. Similar, but different, support will be provided in the next version of Ada ([1], 2.4.2.1). The approach used in Adam is illustrated below:

```
task type WORKER_TYPE;
WORKER: array[I: SUBSCRIPT_RANGE] of WORKER_TYPE;
task body WORKER_TYPE is
   -- the value of I can be used in these declarations
```

```
      ...
   begin
      -- the value of I can be used here
      ...
   end WORKER_TYPE;
```

The base type of such an indexed array must be a task type.

Chapter 2

Rules for Diva Procedures

This chapter presents the rules for declaring and instantiating diva procedures, which can be called from either outside or inside a multiway rendezvous. Rules for diva calls inside a multiway rendezvous are presented in Chapters 3 and 4 and rationales for the Adam constructs are given in Chapter 5.

The formal syntax for a generic diva procedure will be given shortly; roughly speaking, it is a recursive procedure having a restrictive syntax that facilitates translation for parallel computers. A call on an instantiated diva procedure operates on one or more vectors having the same length. The formal parameters corresponding to these vectors are called ***dynamic*** parameters, since the recursive calls of the diva procedure use values of such parameters having progressively shorter lengths. The remaining items in the diva procedure formal parameter list are called ***nondynamic*** parameters. The operation performed by the diva call can either assign a value to one or more nondynamic parameters, assign values to the components of one or more dynamic parameters, or both. For instance, the diva procedure FIND_PRODUCT of Section 1.6 assigns a value to one nondynamic parameter, the diva procedure FIND_LOC of Section 5.1 assigns values to two nondynamic parameters, and the diva procedure UPDATE at the end of Appendix A assigns values to the components of two dynamic parameters.

The formal syntax of a generic diva procedure P is indicated in figures 2.1 and 2.2. The notation { } denotes zero or one of the enclosed items, { }* denotes zero or one or more of the enclosed items, and { }+ denotes

```
generic_diva_procedure_declaration
  ::= GENERIC
        TYPE range_type_mark IS RANGE < > ;
      { TYPE type_mark IS ARRAY ( range_type_mark ) OF type_mark ; }+
      DIVA PROCEDURE generic_diva_procedure_name
        (   dynamic_parameter_declaration { ; dynamic_parameter_declaration }*
            { ; non_dynamic_declarations }  ) IS
      { pure_ada_code1 }
      BEGIN
        IF dynamic_parameter_name ' LENGTH = 1
        THEN
        { pure_ada_code2 }
        ELSE
          generic_diva_procedure_name
             ( dynamic_parameter_name ' INITIAL
               { , dynamic_parameter_name ' INITIAL }*
               { , <identifier> }* ) ;
          generic_diva_procedure_name
             ( dynamic_parameter_name ' FINAL
               { , dynamic_parameter_name ' FINAL }*
               { , <identifier> }* ) ;
        { pure_ada_code3 }
        END IF ;
      { RETURN ; }
      END   generic_diva_procedure_name   ;
```

Figure 2.1: Syntax for Generic Diva Procedure Declaration.

one or more of the enclosed items. An alternative is denoted by |. Each of the following nonterminals produce an identifier: generic_diva_procedure_name, range_type_mark, type_mark, and dynamic_parameter_name. All instances of range_type_mark within P must produce the same identifier. All instances of generic_diva_procedure_name within the diva procedure P must produce the identifier P.

Nonterminals with the prefix pure_ada_code have the following additional static semantic restrictions. The declaration block pure_ada_code1 and the block pure_ada_code_3 cannot access a dynamic parameter. The only attribute of a dynamic parameter that can be accessed within the block pure_ada_code2 is the attribute FIRST and the use of this attribute is the only way a component value of a dynamic parameter can be accessed within the block. Access to nonlocals from within P is prohibited; thus objects used within P, including subprograms, must be declared within P.

```
dynamic_parameter_declaration
  ::=  dynamic_parameter_name { , dynamic_parameter_name }* : mode type_mark

mode
  ::=  IN     |   OUT     |     IN OUT

non_dynamic_declarations
  ::=  in_parameter_declaration
       { ; in_parameter_declaration }*                |
       out_or_in_out_parameter_declaration
       { ; out_or_in_out_parameter_declaration }*  |
       in_parameter_declaration
       { ; in_parameter_declaration }*
         ; out_or_in_out_parameter_declaration
       { ; out_or_in_out_parameter_declaration }*

in_parameter_declaration
  :: = identifier_list : IN type_mark           |
       identifier_list : IN range_type_mark

out_or_in_out_parameter_declaration
  ::=  identifier_list : OUT type_mark          |
       identifier_list : OUT range_type_mark    |
       identifier_list : IN OUT type_mark       |
       identifier_list : IN OUT range_type_mark

identifier_list
  ::=  <identifier> { , <identifier> }*
```

Figure 2.2: Syntax for Diva Procedure Formal Parameters.

The following objects cannot be declared within P: tasks, diva procedures, and diva instantiations. Two new attributes, *INITIAL* and *FINAL*, are defined by the implementation for each dynamic formal parameter in an activation of a diva procedure, access to these attributes is only permitted inside the two recursive calls on the generic diva procedure, and the attributes satisfy the three conditions in Figure 2.3, where A_1, ..., A_m are the dynamic parameters.

Since the syntax rules don't let the programmer specify the division point used for dividing a dynamic parameter A into A'*INITIAL* and A'*FINAL*, the implementation chooses the division point. This key concept of diva procedures is known as the **nondeterministic divide**. For

1. A_i equals A_i'$INITIAL$ concatenated with A_i'$FINAL$.
2. Both A_i'$INITIAL$ and A_i'$FINAL$ have nonempty ranges.
3. The range of A_i'$INITIAL$ equals the range of A_j'$INITIAL$ and the range of A_i'$FINAL$ equals the range of A_j'$FINAL$.

Figure 2.3: Properties of Attributes $INITIAL$ and $FINAL$, for each i and j between 1 and m.

the nonrecursive call, the range of all dynamic parameters is specified by the first generic actual parameter in the instantiation. For the recursive calls, the underlying implementation supplies the constrained range, ensuring that conditions 1 through 3 of Figure 2.3 are satisfied. Condition 3 means the same division point is used for multiple dynamic parameters, which is consistent with the fact that the syntax for a generic diva procedure requires all dynamic parameters to have the same range.

As is true for Ada, the specification of a generic diva procedure can be separate from the declaration of the generic procedure body but the instantiation of the generic procedure cannot precede the declaration of the generic procedure body. The placement of a generic diva procedure instantiation depends on whether calls are made on the instantiated procedure outside or inside a multiway rendezvous:

- **Normal instantiation:** Calls on such an instantiated diva procedure are permitted anywhere outside a multiway rendezvous that an Ada procedure call is permitted. Such an instantiation can be placed anywhere an Ada basic declarative item is permitted.[1]
- **Private instantiation:** Calls on such an instantiated diva procedure are only permitted within a meet statement or multiway accept statement. The instantiation is placed in a private part of a task (or task type) `T`. The declaration of the task is written as follows, for an instantiation `I` of a generic diva procedure `G`, where `A1` and `A2` correspond to generic parameters:

```
task T is
   ... entries are declared here, if there are entries
private
  diva procedure I is new G(A1, A2);
```

[1]See Section 3.9 of [26] for the definition of such an item.

```
    ... additional diva instantiations can be placed here
  end T;
```

The reserved word **type** can appear between **task** and T, when appropriate. If the diva call is in a meet statement within the body of T, then T must be a task type and there must be one and only one array declared to be of type T.[2] If the diva call is in a multiway accept statement within the body of T, there must be an array of tasks each component of which makes calls on the multiway accept. Note that in either case there is an array of tasks that participates in the multiway rendezvous.

[2]The syntax of Ada permits only one array to have a particular task type, if members of the array call other members of the array, since the array name itself must appear in the body of the task type.

Chapter 3

Rules for the Meet Statement

Although a more radical departure from Ada, the meet statement is generally superior to the multiway accept because the rules are simpler, the notation is more concise, and yet the same expressive power is provided for the kinds of applications described in this report. Rationales for the rules in this chapter are given in Chapter 5.

3.1 Meet Statements

The formal syntax of the meet statement is indicated in Figure 3.1, where each of the nonterminals diva_instantiation_name, array_name, and simple_name produces an identifier. Square brackets are not used as metasymbols in the formal syntax; they are lexical units within the Adam language. The meet statement can only appear inside the statement body of a task type `T` that is used in declaring a one-dimensional indexed array of worker tasks, `W[1]`, ..., `W[`m`]`.

The semantics of the meet statement are as follows: when any one of `W[1]`, ..., `W[`m`]` reaches the meet statement it becomes suspended until all have reached the meet statement, whereupon the threads of control of `W[1]`, ..., `W[`m`]` are combined into a single thread of control during the execution of the multiway_rendezvous_statements.

Axioms for **meet** and **end meet** in the style of Hoare[15] are given in Figure 3.2 for programs satisfying the following property: if one of the workers `W[1]`, ..., `W[`m`]` has write access to a nonlocal memory location outside meet statements, no other worker has read access to this memory

```
meet_statement
  ::= MEET multiway_rendezvous_statements END MEET;

multiway_rendezvous_statements
  ::= NULL ; | { diva_call_within_multiway_rendezvous }+

diva_call_within_multiway_rendezvous
  ::= diva_instantiation_name ( actual_parameter_list ) ;

actual_parameter_list
  ::= { array_name , | simple_name [-] . simple_name , }+
      { { simple_name , }* simple_name }
```

Figure 3.1: Syntax for the Meet Statement.

location outside meet statements. This property is easy to satisfy for nearly all Adam programs solving the problems listed in Chapter 1. In Figure 3.2, $P(i)$ and $Q(i)$ are predicates with the free variable i. The range of i is the same as that of the array of tasks and any program variables appearing in the predicates must be visible from the point in the text of W[i] where the predicate appears. In addition, if an array X that is nonlocal to the task type for W appears within either of these predicates, the only subscript value that can be used with X in these predicates is i. Section 5.2 discusses the informal meaning of the two semantic axioms.

3.2 Diva Procedure Calls

A "private" instantiation is required for a diva call within a meet statement, as explained at the end of Chapter 2.

Let W be an array of tasks with range SUBRANGE whose task type T contains a meet statement. Informally, the meet statement defines a meeting among the members of W, where the work of W[i] during the meeting is defined by the diva call within the code for W[i]. Although the code for each member of W is the same, it is possible for such a diva call to involve different nondynamic **in** values for different members of W and different nondynamic **out** values may be received by different members of W. When either or both occur, the effect of the diva call *on the local variables of* W[i] *and on the* i^{th} *component of nonlocal arrays* is exactly the same as if all workers had used the nondynamic **in** values used by the particular worker W[i] and all workers had received the nondynamic **out** values received by the particu-

Axiom for **meet**

$\{ P(i) \}$ **meet** $\{ \forall_i P(i) \}$

Axiom for **end meet;**

$\{ \forall_i Q(i) \}$ **end meet;** $\{ Q(i) \}$

Figure 3.2: Semantic Axioms. Here curly braces have the usual meaning for Hoare semantics.

lar worker `W[i]`. This rule keeps diva procedures relatively easy to write and understand yet enables them to have a powerful effect. The diva procedures `FIRST_RIGHT_GREATER` and `FIND_PLATEAU_LENGTH` in Appendix A illustrate this combination of simplicity and power.

Let `X` be either the task index or a variable declared within `T`, and let `Y` be a nonlocal vector visible from the point of the diva call. Both `W[-].X` and `Y` can be used as an actual parameter corresponding to a dynamic parameter; the former denotes a virtual vector whose range is `SUBRANGE` and whose value at the i^{th} component is equal to `W[i]`'s value of `X`. Such hyphen notation may only be used within diva calls. When `X` is used as an actual parameter corresponding to a nondynamic parameter, it denotes the value of `X` for that particular worker.

Let `I` be an instantiation of a generic diva procedure `G` that has nondynamic **in** parameters `X1`, ..., `Xk` and suppose calls on `I` appear inside meet statements. If the programmer knows that whenever such a call is executed, all values contributed by the participating tasks for the actual corresponding to `X1` are equal, all values corresponding to `X2` are equal, ..., and all values corresponding to `Xk` are equal, the following pragma should be placed after the instantiation of `I` in the private part containing the instantiation

```
pragma SAME_IN_VALUES (I);
```

If the programmer wishes to indicate that pragma `SAME_IN_VALUES` applies to all instantiations of `G` in the program, this can be accomplished using the single pragma

```
    pragma SAME_IN_VALUES (G);
```

inserted after the specification or body of G in the same declaration block as G.

Pragma SAME_IN_VALUES helps the translator generate efficient target code. The presence or absence of the pragma SAME_IN_VALUES affects only the performance of the target code, not its semantic correctness. However, since the effect on performance can be considerable, the presence or absence of such a pragma is reported during the execution of the ADAM command whenever the Adam program has a generic diva procedure G having a non-dynamic **in** parameter such that some instantiation of G is called from within a meet statement. The second program in Appendix A illustrates this pragma.

Chapter 4

Rules for the Multiway Accept

Although the multiway accept is a more straightforward extension of Ada than the meet statement, the rules for the multiway accept are more complex. The rationales for rules in this chapter are given in Chapter 5.

4.1 The Multiway Accept

The syntax for the **multiway accept** statement is indicated in Figure 4.1, where curly braces and the nonterminals multiway_rendezvous_statements and mode are explained in Chapter 2 and Figure 3.1. The remaining nonterminals in Figure 4.1 each produce an identifier. Square brackets are not used as metasymbols in the formal syntax, since they occur as lexical units within Adam.

A multiway accept statement can appear only within a task body or task type body. For a given multiway accept entry_family_name within a given scope, there must be at most one multiway accept statement, and if one exists, there must be one and only one array of tasks that make calls on the multiway accept entry family. Moreover, the multiway accept statement cannot be contained within the tasks of this array, the discrete_range_type_name for the entry family must be the same as that used in declaring the array of tasks, and the task index variable must be the only expression used by a member of the task array in determining which member of the entry family to call.

For a multiway accept entry family `E`, let `SUBSCRIPT_RANGE` be the discrete_range_type_name, whose range is `1..M`. Also let `T` be the task con-

```
multiway_accept_statement
  ::=  ACCEPT   entry_family_name    [ entry_family_index_declaration ]
       { (  parameter_specification  { ; parameter_specification }*  )  }
       { DO  multiway_rendezvous_statements  END  { entry_family_name }  } ; |
       ACCEPT   entry_family_name    [ entry_family_index_declaration ]
       { (  parameter_specification  { ; parameter_specification }*  )  } ;

entry_family_index_declaration
  ::=  entry_family_index_variable    :    discrete_range_type_name

parameter_specification
  ::=  simple_name  { , simple_name }*   :   mode  type_mark
```

Figure 4.1: Syntax for the Multiway Accept Statement.

taining the multiway accept statement S that uses E and let W be the array of tasks indexed by SUBSCRIPT_RANGE that make calls on the entries in the family E. The preceding paragraph implies that calls must be made on the entries E[1], ..., E[M] by the tasks W[1], ..., W[M], respectively, and that no other tasks can make calls on these entries. When each member of the family E has been called and task T has reached the statement S, the multiway_rendezvous_statements within S are executed by T, with all members of the array W suspended during the execution. After the execution of the multiway_rendezvous_statements all M + 1 tasks resume their execution in parallel.

To consider the parameter passing that supports the execution of S, let X be a simple_name used in the parameter_specification. An instance of X exists for each i in the range 1..M. If the mode of X is in or in out, the value of the i^{th} instance of X at the beginning of execution of multiway_rendezvous_statements is the value of the corresponding actual parameter in the entry call made by W[i]. If the mode of X is out or in out, the value of X at the end of execution of multiway_rendezvous_statements is the value received by the corresponding actual parameter of W[i].

The underlying implementation need only be consistent with the above semantics: the ordering of suspension of tasks is nondeterministic and the ordering of unsuspension of tasks is nondeterministic. As indicated in Figure 4.1, a multiway accept need not contain multiway_rendezvous_statements, just as an Ada accept need not have a statement part. Such a multiway accept has the same semantics as a multiway accept containing just a null statement.

No Ada accept can use a multiway accept entry family name and multiway accepts cannot be used to select options in an Ada select statement.

4.2 Syntax for a Multiway Accept Entry Declaration

An entry family used with a multiway accept must be declared with explicit use of the index for the family as in

```
entry E[I: SUBSCRIPT_RANGE] (X: in T1; Y: out T2);
```

The use of the square brackets indicates these entries are multiway accept entries, rather than Ada accept entries. Like Ada entries, the use of a multiway accept entry family in the multiway accept statement must have exactly the same form as in the declaration; this includes the choice of identifiers and the choice of omitting or not omitting an explicit occurence of `in`. In addition, the identifier used in declaring the index of the entry family must be the same as the identifier used in declaring the index of the array of tasks that call the multiway accept and this must be the same as the identifier used as the index in making a call on the multiway accept entry.

4.3 Syntax for a Multiway Accept Entry Call

A task `W[I]` calls the multiway accept entry `E[I]` of task `T` using the notation

```
T.E[I] (...);
```

where `I` is the task index identifier declared for the array `W` and where ... represents the actual parameter list. Each actual parameter must be a local variable of the task type for `W`.

Multiway accept entry calls cannot be used to select options in an Ada select statement.

4.4 Diva Procedure Calls

A "private" instantiation is required for a diva call within a multiway accept statement, as explained at the end of Chapter 2.

Diva calls are used for operating on nonlocal vectors and on the vector of formal parameters within a multiway accept statement. (Diva calls may also be used within a meet statement, as explained in Chapter 3, and outside a multiway rendezvous, as explained in Chapter 2.)

The notation for accessing actual parameters within diva calls of a multiway accept statement will be illustrated for the multiway accept:

```
accept E[I: SUBSCRIPT_RANGE] (X: <mode1> T1; Y: <mode2> T2) do
  multiway_rendezvous_statements
end E;
```

The notation `E[-].X` and `E[-].Y` can be used as an actual parameter corresponding to a dynamic parameter to refer to a virtual vector whose range is `SUBSCRIPT_RANGE` and whose component values are the values of the relevant formal parameters. Thus, whereas within a diva call of a meet statement the use of the notation

```
IDENTIFIER1[-].IDENTIFIER2
```

refers to a virtual vector of *local variables*, this same notation inside a multiway accept statement block refers to a virtual vector of multiway accept *parameter* values.

The actual parameter corresponding to a formal parameter of the diva procedure can refer to a variable declared prior to the multiway accept statement. However, if the formal parameter is a dynamic parameter, the following rule applies: the declaration of the variable used as the actual parameter must be visible at the multiway accept entry calls.

Chapter 5

Rationales for Adam Constructs

Three principles motivate many of the Adam design decisions:

1. **Provide a high level of abstraction.**
2. **Permit a decentralized implementation whenever feasible.**
3. **Don't ask a programmer to make a choice if the choice doesn't matter.**

By a high level of abstraction we mean that the focus of programming should be on what computation is desired, with the translator having the responsibility of deciding how to implement the desired synchronization and communication.

The second principle is referred to in the rest of this report as "the decentralized implementation strategy". Given a suitable computer, much of the work of a multiway rendezvous and of a diva call within it can be performed by the members of the array of tasks that participate in the multiway rendezvous. There are various ways to carry out such a strategy. A diva call outside a multiway rendezvous can also be implemented using a decentralized strategy, by creating an array of tasks invisible to the Adam programmer.

The third principle is to some extent a corollary of the first principle. There are two extremes in which a particular choice wouldn't matter: situations where any choice among the available options is equally valid and situations in which only a single choice is valid, due to constraints elsewhere in the program.

The first extreme is illustrated in sequential programs by a multiway branch statement. Any choice of order in selecting branches of a multiway branch leads to the same logical result and, since the case statement doesn't ask the programmer to make a choice, it separates concerns of efficiency from concerns of logical correctness. This extreme underlies the desirability of supporting nondeterminism in programming languages, such as that studied by Hoare [18]. Adam supports this kind of nondeterminism by not asking a programmer to make a choice in such things as the order of combining within the computation of a diva call and the order in which required calls on a multiway accept are processed.

The second extreme is illustrated in Adam by the fact that the programmer uses hyphen notation rather than indicating the range of a virtual vector used by a diva call within a multiway rendezvous. The only valid choice of this range is the range of the tasks participating in the multiway rendezvous.

Exceptions are made to the third principle when necessary to keep Adam compatible with Ada. For example, Ada requires that the choice of identifiers in an accept statement be the same as the choice in the corresponding entry declaration and the Adam language has a similar requirement.

Rather than striving for maximal generality, the Adam language demonstrates how the multiway rendezvous, combined with a suitably restrictive procedure employing no explicit parallelism, can provide a simple level of abstraction yet support translation to target code appropriate for a range of parallel computers. There are certainly straightforward ways in which the Adam constructs can be generalized.

5.1 Diva Procedures

The decentralized implementation strategy distributes the work of each call on a diva procedure. Thus diva procedures facilitate a time-efficient implementation, yet permit the programmer to think at a high level of abstraction.

When appropriate for the target machine, the target Ada code produced by the Adam translator uses the following facts:

- Given sufficiently many processors, a diva procedure can be executed in time proportional to $log(n)$, where n is the length of the vectors used as input by the procedure. This can be accomplished by implementing the nondeterministic divide as a divide at the middle of a vector and letting multiple processors use a binary combining tree to share in the computation of the procedure.

- The nondeterministic divide can be implemented as a divide just before the last component of a vector (or just after the first component) thereby yielding a sequential loop. Such a loop is typically more efficient than recursion for implementing a diva procedure on a sequential computer. Thus during a single use of a diva procedure, some of the recursive activations — such as those near the beginning of execution of the procedure — can be carried out in parallel and the rest can be implemented sequentially without the overhead of recursive calls. This facilitates an implementation in which processors do some of the work on the procedure independently and then combine their partial results. If the divide were defined to be a deterministic divide at the middle, an implementation using a sequential loop (without a stack) would not be possible, in general.

- The nondeterministic divide need not be implemented the same way on different processors, even for a single diva call within a single parallel computer. For example, consider a `RING` target machine having an even number p of processors. When the pragma `SAME_IN_VALUES` is not used, the target code produced by the Adam translator implements the nondeterministic divide so the number of ways one vector of length p is decomposed into single components at the same time by the different processors is $p/2$. It is even possible to implement the nondeterministic divide so that — at the same time — each processor uses a different decomposition.

The correctness of a well-written diva procedure can be proven in a straightforward way. We illustrate this using the diva procedure in Figure 5.1. As explained in Chapter 2, the underlying implementation chooses A'INITIAL and A'FINAL, used in making the recursive calls, to be nonempty slices of A such that A is A'INITIAL concatenated with A'FINAL. The programmer is responsible for ensuring the correctness of such a diva procedure regardless of how the implementation chooses to make such a nondeterministic divide.

Such a proof of correctness, based on strong induction, can proceed as follows: First show that, when A has length 1, the diva procedure satisfies the specification given in its comments. Then assume the length of A is greater than 1 and the diva procedure satisfies its specification for all smaller vectors than the length of A, so that

`INITIAL_MAX` is assigned the maximum value of A'INITIAL,
`INITIAL_LOC` is assigned the first location of the maximum of A'INITIAL,
`FINAL_MAX` is assigned the maximum value of A'FINAL, and
`FINAL_LOC` is assigned the first location of the maximum of A'FINAL.

```
generic
  type RANGE_TYPE is range <>;
  type DYNAMIC_VECTOR is array (RANGE_TYPE) of INTEGER;
diva procedure FIND_LOC (A: in DYNAMIC_VECTOR;
                         MAX: out INTEGER;
                         LOC: out RANGE_TYPE) is
  -- MAX is assigned the maximum value of A.
  -- LOC is assigned the first location of the maximum of A.
  INITIAL_MAX, FINAL_MAX,
  INITIAL_LOC, FINAL_LOC: INTEGER;
begin
  if A'LENGTH = 1 then
    MAX := A(A'FIRST);
    LOC := A'FIRST;
  else
    FIND_LOC (A'INITIAL, INITIAL_MAX, INITIAL_LOC);
    FIND_LOC (A'FINAL,   FINAL_MAX, FINAL_LOC);
    if INITIAL_MAX >= FINAL_MAX then
       MAX := INITIAL_MAX;
       LOC := INITIAL_LOC;
    else
       MAX := FINAL_MAX;
       LOC := FINAL_LOC;
    end if;
  end if;
end FIND_LOC;
```

Figure 5.1: Generic Diva Procedure to Find the First Location of the Maximum of an Integer Valued Vector.

To complete the proof, simply show that the false branch of the `if` statement assigns the specified values to `MAX` and `LOC`.

A diva procedure can abstract away issues irrelevant to the logical correctness of the procedure in question. Such issues include whether a loop will be used for sequential implementation, whether parallelism will be used to implement the procedure, whether the target computer is MIMD or SIMD, whether processors will do some of the combining work independently before combining their partial results, and what technique will be used in combining partial results from different processors (a combining tree of processors, left or right shifts, pointer doubling, or other technique). The resulting high-level syntax results in code more portable and easier to read, write, and verify than approaches failing to abstract away such details.

The syntax rules of a generic diva procedure P are compiler-enforceable. The requirement that subprograms used within P be declared within P permits a compiler to disallow additional recursive calls on P as a result of executing this Ada code. The requirement that a diva instantiation cannot appear within the declaration of a diva procedure simplifies the implementation (and we have not yet found important applications of such a hierarchical approach except for computing reductions of multidimensional arrays).

The syntax of generic diva procedures requires that dynamic parameters be listed first, followed by nondynamic parameters of mode `in`, if any, followed by other nondynamic parameters, if any. These restrictions enforce a convention intended to make the language easier for programmers to use. For instance, since a diva call operates on vectors, listing such vectors first in the diva call places them in a prominent position. Since they require the programmer to remember additional rules, the simplicity provided by these restrictions is debatable, and they might be lifted in the future. Only additional experience can resolve such issues. (A similar approach was taken by Ada in requiring that basic declarations precede later declarations; experience with this restriction has led to eliminating the restriction in the next version of Ada.[1])

Although an instantiation of an Ada generic procedure can be either a basic or a later declarative item, a normal instantiation of a generic diva procedure must be a basic declarative item. This restriction allows the translator to insert (at the corresponding place in the target Ada code) items permitted as basic, but not as later, declarative items. This restriction and the fact that such an instantiation cannot precede the declaration of the generic procedure body imply that an Adam program must have more than a single declaration block.

The private part of a task type specification is an extension of Ada that provides information to a task of the task type. Like the private part of a package specification, information within the private part of a task type specification is not visible outside the task type itself. Placing diva procedure instantiations within a private part of a task type specification is natural since an instantiated diva procedure can be viewed as a private resource. In addition, the translator can insert (at the corresponding place in the target Ada code) entries to support the decentralized implementation strategy, with assurance that type names will bind correctly to the appropriate meanings.

5.2 Meet Statements

Notice that, placed in the code for the task type of an array of workers, the lines

```
meet
  null;
end meet;
```

behave as a barrier [14], ensuring all workers synchronize before performing further work.

Informally, the first semantic axiom in Figure 3.2 states that if $P(i)$ holds prior to **meet**, then $\forall_i P(i)$ holds immediately after **meet** in the program text. This axiom conveys in a formal way the informal notion that no worker can get beyond **meet** until it and all other workers in the array have arrived at **meet**. Informally, the second semantic axiom states that if $\forall_i Q(i)$ holds before **end meet**, then $Q(i)$ holds immediately afterwards in the program text. This axiom is based on the restriction required by these axioms, that if one of the workers has write access to a nonlocal memory location outside meet statements, no other worker has read access to this memory location outside meet statements. The reason the full strength of $\forall_i Q(i)$ cannot be asserted in the program text immediately after **end meet** is because this could create a race condition: for any i_1, worker i_1 could modify the truth value of $Q(i_1)$, such as by executing a statement that assigns a value to the ${i_1}^{th}$ component of a nonlocal array.

To illustrate the use of these semantic axioms, consider the second version of `FACT_.ADAM` in Section 1.6. In this example, $n = 5$ and $P(i)$ can be the predicate

`WORKER[`i`]`'s `MY_VALUE` $= i$

and $Q(i)$ can be the predicate

WORKER[i]'s MY_RESULT = $\prod_{i \leq n} i$

where $\prod$ denotes a product.

The decentralized implementation strategy lets the work of a meet statement be performed by the members of the array of tasks whose task type contains the meet statement.

The Adam translator relies exclusively on the pragma SAME_IN_VALUES for the information it provides. While it is conceivable that a compiler could extract this information for the applications described in this report, such information cannot be extracted for all Adam programs. For instance, input data could be used in the task type of an array of workers to determine the value to assign to a local variable used as an actual parameter corresponding to a nondynamic **in** parameter in a diva call within a meet statement.

5.3 Square Bracket Notation

Square brackets are used in the Adam language as a consistent notational device that

- Avoids the use of reserved words, but makes clear an extension to Ada is being used. The extensions to Ada, except for meet statements and diva procedure declarations, instantiations, and calls, are:

  - multiway accept entry declarations
  - multiway accept entry calls
  - multiway accept statements
  - indexed task array declarations

  No new reserved words are used in these extensions. Instead square brackets indicate the difference between Adam and Ada.

- Suggests powerful constructs are being used. For example, the Ada code needed to ensure that the members of an array of n tasks gain automatic access to their index requires the time for executing at least O($log(n)$) rendezvous.

Furthermore, if rounded parentheses were used in a multiway accept entry call, then for a task T the syntax T.E(I) would be ambiguous unless one looked at declarations appearing possibly far earlier in the program.

It could either mean to call the entry `E` of an Ada accept (with actual parameter `I`) or to call the parameterless member `E(I)` of a multiway accept statement. Wirth has recommended that the syntax for a programming language should be capable of recursive descent parsing with one-symbol lookahead, regardless of how parsing is actually accomplished. He argues that this syntax is easy for programmers to understand.[1]

5.4 Multiway Accepts

The decentralized implementation strategy is the reason why a non-hyphened variable used as an actual parameter corresponding to a dynamic parameter in a diva call within a multiway accept must be visible at the multiway accept entry call.

Requiring that only a single multiway accept appear in a task for each entry family and prohibiting its use within a select statement make it easy for the implementation to determine which entry calls are associated with which multiway accepts in support of the decentralized implementation strategy.

In the Ada accept statement the entry accepted must have exactly the same form as the declaration of the entry. The same holds for the multiway accept in Adam for compatibility between the two languages. This requires that the index of an entry family appear in the declaration of the entry family. Restrictions on the use of identifiers (in declaring the indexes of an entry family, in declaring the array of tasks, and in making entry calls) ensure that each task in the array of tasks can only call the corresponding entry of the multiway accept family. These restrictions not only simplify the Adam translator but are consistent with the use of reasonable structure in an Adam program

A dotted notation, or something similar, is needed to create a virtual vector using the formal parameters of a multiway accept. This facility is to be contrasted with a restriction in Section 4.1.3, Paragraph 17, of the Ada reference manual [26]. Such notation facilitates the decentralized implementation strategy.

5.5 Comparison of Two Approaches

The meet statement and the multiway accept statement provide two different methods of supporting the multiway rendezvous within an Adam

[1]See page 164 of Wirth's Turing award lecture [28] and page 29 of the reprinted version of [27].

program. A conceptual difference is that the multiway accept is used with an array of tasks and a centralizing task, which can execute the multiway accept statement as just one of many of its activities, whereas the meet is used with just an array of tasks.

Comparisons of programs like those mentioned in Chapter 1 (and the two listed in that chapter that use the multiway rendezvous) suggest that the meet statement leads to shorter and simpler programs. The rules of the meet statement are also less awkward than the rules of the multiway accept. Awkward rules for the multiway accept, for which there are no corresponding rules for the meet statement include: the identifier used in declaring the index of a multiway accept entry family must be the same as the identifier used in declaring the array of tasks that call the multiway accept and this must be the same as the identifier used as the index in making a call on the multiway accept, neither a multiway accept nor a multiway accept entry call can be used to select options in an Ada select statement and, when hyphen notation is not used to refer to a dynamic parameter in a diva call within a multiway accept, the variable used for this purpose must be visible at the multiway accept entry calls.

References

[1] Ada 9X Mapping/Revision Team, Draft Ada 9X Project Report, Ada 9X Mapping Document, vol I, Office of the Under Secretary of Defense for Acquisition, Washington, DC, March 1992.

[2] Andrews, G.R. Synchronizing resources. ACM Transactions on Programming Languages and Systems, 3, 4 (Oct. 1981), 405-430.

[3] Banatre, J.-P. and Le Metayer, D. Programming by multiset transformation, Comm. ACM, Jan. 1993, 98-111.

[4] Brinch-Hansen, P. Distributed processes: a concurrent programming concept. Comm. ACM, 21, 11 (Nov. 1978), 934-941.

[5] Charlesworth, A. Synchronization and Communication Using Compacts. M.S. Thesis, Computer Science Department, U. of Virginia, Charlottesville (Aug. 1983).

[6] Charlesworth, A. The multiway rendezvous. ACM Transactions on Programming Languages and Systems, 9, 2, (July 1987), 350-366.

[7] Charlesworth, A. On a class of divide and conquer algorithms. Preliminary report. Abstracts of the Amer. Math. Soc., 10, 6 (Nov. 1989), 492.

[8] Charlesworth, A. The nondeterministic divide, Tech. Rep. IPC-TR-90-005, Inst. for Parallel Computation, U. of Virginia, Charlottesville, Nov. 1990.

[9] Charlesworth, A. A characterization of associativity, Tech. Rep. IPC-TR-90-007, Inst. for Parallel Computation, U. of Virginia, Charlottesville, Nov. 1990.

[10] Charlesworth, A. Placing a diva call within either a multiway rendezvous or a parallel for loop. Preliminary report. Abstracts of the Amer. Math. Soc., 14, 6 (Oct. 1993), 705.

[11] Deo, N., Pang, C.Y., and Lord, R.E. Two parallel algorithms for shortest path problems. Proc. of the 1980 Int. Conf. on Parallel Processing, 244-253.

[12] Deo, N., and Yoo, Y.B. Parallel algorithms for the minimum spanning tree problem. Proc. of the 1981 Int. Conf. on Parallel Processing, 188-189.

[13] Fox, G. et. alia. *Solving Problems on Concurrent Processors: Vol 1 General Techniques and Regular Problems*, Prentice-Hall, Englewood Cliffs, N.J., 1988.

[14] Jordan, H. J. Programming language concepts for multiprocessors. Parallel Computing, 8 (1988), 31-40.

[15] Hoare, C. A. R. and Wirth, N. An axiomatic definition of the programming language Pascal. Acta Informatica, 2 (1973), 335-355

[16] Hoare, C. A. R. Communicating sequential processes. Comm. ACM, 21, 8 (Aug. 1978), 666-677.

[17] Hoare, C. A. R. The emperor's old clothes. Comm. ACM, 24, 2 (Feb. 1981), 75-83.

[18] Hoare, C. A. R. *Communicating Sequential Processes* Prentice-Hall Intern., Englewood Cliffs, N.J., 1985.

[19] Hockney, R. W. and Jesshope, C. R. *Parallel Computers: Architecture, Programming and Algorithms*. Higer, London, U. K., 1981.

[20] Moe, Z. G., *A Formal Model for Divide-and-Conquer and its Parallel Realization*, Ph.D. thesis, Yale University, May 1990.

[21] Nitzberg, B. and Lo, V., Distributed shared memory: a survey of issues and algorithms, IEEE Computer, (Aug. 1991), 52-60.

[22] NSF. The National Science Foundation Metacenter. A report prepared for the program advisory committee to the National Science Foundation Division of Advanced Scientific Computing, February 1992.

[23] Pancake, C. M. Software support for parallel computing: Where are we headed? Comm. ACM, 34, 11, (Nov. 1991), 52-64.

[24] Quinn, M.J. The Design and Analysis of Algorithms and Data Structures for the Efficient Solution of Graph Theoretic Problems on MIMD Computers. Ph. D. Dissertation, Dep. Computer Science, Wash. State Univ., Pullman, Wash., 1983.

[25] Steele, G. S., Jr. ***Data Parallel Algorithms***, Vol. 3 of Distinguished Lecture Series, University Video Communications, 1991.

[26] United States Department of Defense. ***Reference Manual for the Ada*** *Programming Language.* ANSI/MIL-STD-1815A-1983, American National Standards Institute, 1983.

[27] Wirth, N. On the design of programming languages. Proc. IFIP Congress 74, North-Holland Pub. Co., 386-393. Reprinted as pages 23-30 of E. Horowitz, ed. *Programming Languages: A Grand Tour*, Computer Science Press, Rockville MD, 1983.

[28] Wirth, N. From programming language design to computer construction, Comm. ACM, 28, 2 (Feb. 1985), 160-164.

Appendix A

Additional Examples

The following examples are given here:

- USE_FIRST_RIGHT_GREATER_.ADAM, which computes, for each position in a vector, the first position to the right holding a greater value, if there is such a position,
- USE_LOOKUP_.ADAM, which computes the position in a not-necessarily-sorted vector whose component equals a given key value,
- USE_FIND_PLATEAU_LENGTH_.ADAM, which computes, for each position in a vector, the number of contiguous positions having the same value,
- SORT_.ADAM, which sorts a vector, and
- USE_PREFIX_.ADAM, which computes a parallel prefix result,

as well as a diva procedure, UPDATE, which assigns values to the components of two vectors. An Ada package PARALLEL_OUT is used (but not shown) that permits a task to build up an output buffer via calls on PRINT. The task can eventually execute a PRINT_LINE call to print the buffer, thus avoiding unreadable interleaving of output from members of an array of tasks.

A.1 USE_FIRST_RIGHT_GREATER_.ADAM

This first example also illustrates what a VAX/VMS terminal session can look like. To see that that the diva procedure FIRST_RIGHT_GREATER satisfies the specifications in its comment block, use strong induction as illustrated in Section 5.4, and assume that the parameters GIVEN_LOC and

GIVEN_VALUE are two *fixed* values. The rule given in the second paragraph of Section 3.2 then assures us that the diva call within the meet statement will have the desired effect, even though different members of WORKER actually use *different* values for these two parameters.

```
$ TYPE USE_FIRST_RIGHT_GREATER_.ADAM
with INTEGER_TEXT_IO, TEXT_IO, PARALLEL_OUT;
use INTEGER_TEXT_IO, TEXT_IO;

procedure USE_FIRST_RIGHT_GREATER is

  N: INTEGER;

  type ANS_TYPE is record
     LOC,
     VALUE: INTEGER;
     FOUND: BOOLEAN;
  end record;

  generic
    type RANGE_TYPE is range <>;
    type DYNAMIC_VECTOR is array(RANGE_TYPE) of INTEGER;
  diva procedure FIRST_RIGHT_GREATER (A: in DYNAMIC_VECTOR;
                                      GIVEN_LOC: in INTEGER;
                                      GIVEN_VALUE: in INTEGER;
                                      ANS: out ANS_TYPE) is
    -- ANS.FOUND is true if and only if A has a component to the
    --   right of A(GIVEN_LOC) greater than GIVEN_VALUE.
    -- If ANS.FOUND is true, then:
    --   ANS.LOC is the location of the first value to the right of
    --     A(GIVEN_LOC) greater than GIVEN_VALUE and
    --   ANS.VALUE is A(ANS.LOC).
    L, R: ANS_TYPE;
  begin
     if A'LENGTH = 1 then
        if A'FIRST > GIVEN_LOC and then A(A'FIRST) > GIVEN_VALUE then
           ANS.LOC := A'FIRST;
           ANS.VALUE := A(A'FIRST);
           ANS.FOUND := TRUE;
        else
           ANS.FOUND := FALSE;
        end if;
     else
        FIRST_RIGHT_GREATER (A'INITIAL, GIVEN_LOC, GIVEN_VALUE, L);
        FIRST_RIGHT_GREATER (A'FINAL,   GIVEN_LOC, GIVEN_VALUE, R);
        if L.FOUND then
```

```
          ANS := L;
       elsif R.FOUND then
          ANS := R;
       else
          ANS.FOUND := FALSE;
       end if;
    end if;
 end FIRST_RIGHT_GREATER;

begin -- USE_FIRST_RIGHT_GREATER
  GET (N);
  PUT ("The length of the vector is ");
  PUT (N, 1); NEW_LINE;
  declare
    subtype SUBRANGE is INTEGER range 1..N;
    type VECTOR is array(SUBRANGE) of INTEGER;
    diva procedure MY_FIRST_RIGHT_GREATER is
                   new FIRST_RIGHT_GREATER (SUBRANGE, VECTOR);
    X: VECTOR;
    ANSWER: ANS_TYPE;

    procedure DO_FIRST_RIGHT_GREATER is

      task type WORKER_TYPE is
      private
        diva procedure MY_FIRST_RIGHT_GREATER is new
                       FIRST_RIGHT_GREATER (SUBRANGE, VECTOR);
      end WORKER_TYPE;

      WORKER: array[I: SUBRANGE] of WORKER_TYPE;

      task body WORKER_TYPE is
        MY_ANSWER: ANS_TYPE;
        MY_X: INTEGER;
        package PAR_OUT is new PARALLEL_OUT;
        use PAR_OUT;
      begin
        MY_X := X(I);
        meet
          MY_FIRST_RIGHT_GREATER (X, I, MY_X, MY_ANSWER);
        end meet;
        if MY_ANSWER.FOUND then
          PRINT ("The answer for WORKER["); PRINT(I);
          PRINT ("] is WORKER["); PRINT(MY_ANSWER.LOC); PRINT ("]");
        else
```

```
          PRINT ("No worker to the right of WORKER["); PRINT(I);
          PRINT ("] has a greater value");
        end if;
        PRINT_LINE;
      end WORKER_TYPE;

    begin -- DO_FIRST_RIGHT_GREATER
      null;  -- Activate the array of workers.
    end DO_FIRST_RIGHT_GREATER;

  begin -- declare block
    for I in SUBRANGE loop
      GET (X(I));
    end loop;
    PUT_LINE ("The vector of values is:");
    for I in SUBRANGE loop
      PUT (X(I), 1); NEW_LINE;
    end loop;
    DO_FIRST_RIGHT_GREATER;
  end;  -- declare block
end USE_FIRST_RIGHT_GREATER;
$ ADAM/MACHINE=RING USE_FIRST_RIGHT_GREATER_.ADAM

 WARNING: Pragma SAME_IN_VALUES is not found for diva procedure
          FIRST_RIGHT_GREATER
          Thus any single occurrence of a diva call on an instantiation of this
          diva procedure, where the call occurs within a meet statement,
          is assumed NOT to be made with the same corresponding in values for
          all tasks participating in the call.  If this pragma was omitted
          unintentionally, then the computation of each diva call on this diva
          procedure is likely to take more run-time than necessary.


$ ADA USE_FIRST_RIGHT_GREATER_.ADA
$ ACS LINK USE_FIRST_RIGHT_GREATER
%ACS-I-CL_LINKING, Invoking the VMS Linker for VAX_VMS target
$ ASSIGN USE_FIRST_RIGHT_GREATER.DAT1 ADA$INPUT
$ RUN USE_FIRST_RIGHT_GREATER.EXE
The length of the vector is 10
The vector of values is:
1
2
3
4
5
```

```
2
9
6
5
8
No worker to the right of WORKER[ 10] has a greater value
The answer for WORKER[ 2] is WORKER[ 3]
The answer for WORKER[ 4] is WORKER[ 5]
The answer for WORKER[ 6] is WORKER[ 7]
The answer for WORKER[ 8] is WORKER[ 10]
The answer for WORKER[ 9] is WORKER[ 10]
The answer for WORKER[ 1] is WORKER[ 2]
The answer for WORKER[ 3] is WORKER[ 4]
The answer for WORKER[ 5] is WORKER[ 7]
No worker to the right of WORKER[ 7] has a greater value
$ ASSIGN USE_FIRST_RIGHT_GREATER.DAT2 ADA$INPUT
%DCL-I-SUPERSEDE, previous value of ADA$INPUT has been superseded
$ RUN USE_FIRST_RIGHT_GREATER.EXE
The length of the vector is 10
The vector of values is:
5
4
3
2
1
10
9
8
7
6
No worker to the right of WORKER[ 10] has a greater value
The answer for WORKER[ 2] is WORKER[ 6]
The answer for WORKER[ 4] is WORKER[ 6]
No worker to the right of WORKER[ 6] has a greater value
No worker to the right of WORKER[ 8] has a greater value
No worker to the right of WORKER[ 9] has a greater value
The answer for WORKER[ 1] is WORKER[ 6]
The answer for WORKER[ 3] is WORKER[ 6]
The answer for WORKER[ 5] is WORKER[ 6]
No worker to the right of WORKER[ 7] has a greater value
```

A.2 USE_LOOKUP_.ADAM

The program USE_FIRST_RIGHT_GREATER_.ADAM does not use either of the pragmas

    pragma SAME_IN_VALUES (USE_FIRST_RIGHT_GREATER);

or

    pragma SAME_IN_VALUES (FIRST_RIGHT_GREATER);

since neither USE_FIRST_RIGHT_GREATER nor any instantiation of the generic diva procedure FIRST_RIGHT_GREATER is intended to be used with just a single value for GIVEN_LOC. The various WORKER tasks certainly do not use the same value in the diva call

    MY_FIRST_RIGHT_GREATER (X, I, MY_X, MY_ANSWER);

for the actual parameter I corresponding to the parameter GIVEN_LOC (or even for the actual parameter MY_X corresponding to GIVEN_VALUE).

The following simple Adam program illustrates a situation in which this pragma should be used. The generic diva procedure LOOKUP in this program returns the component value of the vector B corresponding to the first position in the vector A where a component value equals a given key, if there is such a component of A. Since the programmer knows the same key value will be used in any particular call on an instantiation of LOOKUP, the pragma SAME_IN_VALUES is used to help the translator generate efficient code for the diva call on MY_LOOKUP.

```
with TEXT_IO, PARALLEL_OUT;
use TEXT_IO;
procedure USE_LOOKUP is

  package INTEGER_TEXT_IO is new TEXT_IO.INTEGER_IO (INTEGER);
  use INTEGER_TEXT_IO;

  N: INTEGER;

  type ANSWER_TYPE is record
    FOUND: BOOLEAN;
    VALUE: INTEGER;
  end record;

  generic
    type RANGE_TYPE is range <>;
    type DYNAMIC_VECTOR is array(RANGE_TYPE) of INTEGER;
  diva procedure LOOKUP (A, B: in DYNAMIC_VECTOR;
                         KEY: in INTEGER; ANS: out ANSWER_TYPE) is
    -- ANS.FOUND is assigned the value TRUE if and only if KEY equals the value
    --   of a component of A.
    -- If ANS.FOUND then ANS.VALUE equals the component value of B
    --   corresponding to the first such component value of A.
    L, R: ANSWER_TYPE;  -- results of left and right slices

  begin
    if A'LENGTH = 1 then
      ANS.FOUND := (A(A'FIRST) = KEY);
      ANS.VALUE := B(A'FIRST);
    else
      LOOKUP (A'INITIAL, B'INITIAL, KEY, L);
      LOOKUP (A'FINAL,   B'FINAL,   KEY, R);
      if L.FOUND then
        ANS := L;
      else
        ANS := R;
      end if;
    end if;
  end LOOKUP;

  pragma SAME_IN_VALUES (LOOKUP);


begin -- USE_LOOKUP
  GET (N);
  PUT ("The length of the vector is ");
  PUT (N, 1); NEW_LINE;
  declare
    subtype SUBRANGE is INTEGER range 1..N;
    type VECTOR is array(SUBRANGE) of INTEGER;
    X, Y: VECTOR;
    KEY: INTEGER;

    procedure DO_LOOKUP is

      task type WORKER_TYPE is
      private
        diva procedure MY_LOOKUP is new LOOKUP (SUBRANGE, VECTOR);
      end WORKER_TYPE;

      WORKER: array[MY_INDEX: SUBRANGE] of WORKER_TYPE;

      task body WORKER_TYPE is
        MY_ANSWER: ANSWER_TYPE;
        package PAR_OUT is new PARALLEL_OUT;
        use PAR_OUT;
      begin
```

```
        meet
          MY_LOOKUP (X, Y, KEY, MY_ANSWER);
        end meet;
        if MY_ANSWER.FOUND then
          PRINT ("WORKER["); PRINT (MY_INDEX); PRINT("] finds the answer");
          PRINT (MY_ANSWER.VALUE); PRINT_LINE;
        else
          PRINT ("WORKER["); PRINT (MY_INDEX); PRINT("] finds no answer");
          PRINT_LINE;
        end if;
      end WORKER_TYPE;

    begin -- DO_LOOKUP
      null;  -- Activate the array of workers.
    end DO_LOOKUP;

  begin
    for I in SUBRANGE loop
      GET (X(I));  GET (Y(I));
    end loop;
    PUT_LINE ("The vectors of keys and values are:");
    for I in SUBRANGE loop
      PUT (X(I), 1); PUT (Y(I)); NEW_LINE;
    end loop;
    GET (KEY);
    PUT ("The key to search for is ");
    PUT (KEY, 1); NEW_LINE;
    DO_LOOKUP;
  end;
end USE_LOOKUP;
```

A.3 USE_FIND_PLATEAU_LENGTH_.ADAM

The Adam program in this section computes, for each position in a vector, the number of contiguous positions having the same value.

```
with INTEGER_TEXT_IO, TEXT_IO, PARALLEL_OUT;
use INTEGER_TEXT_IO, TEXT_IO;

procedure USE_FIND_PLATEAU_LENGTH is

---------------------------------------------------------------------------
--  Each worker finds out how many workers contiguous to it have the
--  same value as the worker itself.
--  We use the phrase "the plateau of A containing A(GIVEN_LOC)" in our
-- comments to mean: "the longest slice of A that both contains the
-- component of A with subscript GIVEN_LOC and has all component values
-- equal to A(GIVEN_LOC)".
-- [This problem appears in Gries, D., The Science of Programming,
-- Springer-Verlag, New York, 1981.]
----------------------------------------------------------------------------

  N: INTEGER;

  type ANS_TYPE is record
     PLATEAU_LENGTH,
     LENGTH,
     FIRST_COUNT,
     LAST_COUNT:             INTEGER;
     PLATEAU_FROM_START,
     PLATEAU_TO_END:      BOOLEAN;
  end record;

  generic
    type RANGE_TYPE is range <>;
    type DYNAMIC_VECTOR is array(RANGE_TYPE) of INTEGER;
  diva procedure FIND_PLATEAU_LENGTH (A: in DYNAMIC_VECTOR;
                                      GIVEN_LOC: in RANGE_TYPE;
                                      GIVEN_VALUE: in INTEGER;
                                      ANS: in out ANS_TYPE) is
    -- If GIVEN_LOC is in the subscript range of A and
    --    A(GIVEN_LOC) = GIVEN_VALUE
    -- then:
    --   ANS.PLATEAU_LENGTH is the length of the plateau of A containing
    --     A(GIVEN_LOC),
    --   ANS.PLATEAU_FROM_START if and only if the plateau of A containing
    --     A(GIVEN_LOC) includes the first component value of A, and
    --   ANS.PLATEAU_TO_END if and only if the plateau of A containing
    --     A(GIVEN_LOC) includes the final component value of A,
    -- otherwise:
    --   ANS.PLATEAU_LENGTH has the value 0,
    --   ANS.PLATEAU_FROM_START is FALSE, and
    --   ANS.PLATEAU_TO_END     is FALSE.
    -- ANS.LENGTH is the length of A.
    -- ANS.FIRST_COUNT is the number of consecutive components of A,
    --   starting with the first component of A, whose value is GIVEN_VALUE.
    -- ANS.LAST_COUNT is the number of consecutive components of A,
    --   ending with the last component of A, whose value is GIVEN_VALUE.
    L, R: ANS_TYPE;
  begin
```

```
if A'LENGTH = 1 then
   ANS.LENGTH := 1;
   if A'FIRST = GIVEN_LOC then
      ANS.PLATEAU_LENGTH := 1;
      ANS.PLATEAU_FROM_START := TRUE;
      ANS.PLATEAU_TO_END := TRUE;
      ANS.FIRST_COUNT := 1;
      ANS.LAST_COUNT := 1;
   else
      ANS.PLATEAU_LENGTH := 0;
      ANS.PLATEAU_FROM_START := FALSE;
      ANS.PLATEAU_TO_END := FALSE;
      if A(A'FIRST) = GIVEN_VALUE then
         ANS.FIRST_COUNT := 1;
         ANS.LAST_COUNT := 1;
      else
         ANS.FIRST_COUNT := 0;
         ANS.LAST_COUNT := 0;
      end if;
   end if;
else
   FIND_PLATEAU_LENGTH (A'INITIAL, GIVEN_LOC, GIVEN_VALUE, L);
   FIND_PLATEAU_LENGTH (A'FINAL,   GIVEN_LOC, GIVEN_VALUE, R);
   if L.PLATEAU_LENGTH > 0 then
      if L.PLATEAU_TO_END then
         ANS.PLATEAU_LENGTH := L.PLATEAU_LENGTH + R.FIRST_COUNT;
      else
         ANS.PLATEAU_LENGTH := L.PLATEAU_LENGTH;
      end if;
   elsif R.PLATEAU_LENGTH > 0 then
      if R.PLATEAU_FROM_START then
         ANS.PLATEAU_LENGTH := L.LAST_COUNT + R.PLATEAU_LENGTH;
      else
         ANS.PLATEAU_LENGTH := R.PLATEAU_LENGTH;
      end if;
   else
      ANS.PLATEAU_LENGTH := 0;
   end if;
   ANS.LENGTH := L.LENGTH + R.LENGTH;
   ANS.PLATEAU_FROM_START := L.PLATEAU_FROM_START or
                (L.LAST_COUNT = L.LENGTH and R.PLATEAU_FROM_START);
   ANS.PLATEAU_TO_END := R.PLATEAU_TO_END or
                (R.FIRST_COUNT = R.LENGTH and L.PLATEAU_TO_END);
   if L.FIRST_COUNT = L.LENGTH then
      ANS.FIRST_COUNT := L.LENGTH + R.FIRST_COUNT;
```

```
        else
           ANS.FIRST_COUNT := L.FIRST_COUNT;
        end if;
        if R.LAST_COUNT = R.LENGTH then
           ANS.LAST_COUNT := L.LAST_COUNT + R.LENGTH;
        else
           ANS.LAST_COUNT := R.LAST_COUNT;
        end if;
     end if;
  end FIND_PLATEAU_LENGTH;

begin -- USE_FIND_PLATEAU_LENGTH
  GET (N);
  PUT ("The length of the vector is ");
  PUT (N, 1); NEW_LINE;
  declare
    subtype SUBRANGE is INTEGER range 1..N;
    type VECTOR is array(SUBRANGE) of INTEGER;
    X: VECTOR;
    ANSWER: ANS_TYPE;

    procedure DO_FIND_PLATEAU_LENGTH is

      task type WORKER_TYPE is
      private
        diva procedure MY_FIND_PLATEAU_LENGTH is
          new FIND_PLATEAU_LENGTH (SUBRANGE, VECTOR);
      end WORKER_TYPE;

      WORKER: array[I: SUBRANGE] of WORKER_TYPE;

      task body WORKER_TYPE is
        MY_ANSWER: ANS_TYPE;
        X_I: INTEGER;
        package PAR_OUT is new PARALLEL_OUT;
        use PAR_OUT;
      begin
        X_I := X(I);
        meet
          MY_FIND_PLATEAU_LENGTH (X, I, X_I, MY_ANSWER);
        end meet;
        PRINT ("The plateau for WORKER["); PRINT(I); PRINT ("] has length ");
        PRINT(MY_ANSWER.PLATEAU_LENGTH); PRINT_LINE;
      end WORKER_TYPE;
    begin -- DO_FIND_PLATEAU_LENGTH
      null;  -- activate the array of workers
    end DO_FIND_PLATEAU_LENGTH;

  begin -- declare block
    for I in SUBRANGE loop
      GET (X(I));
    end loop;
    PUT_LINE ("The vector is:");
    for I in SUBRANGE loop
      PUT (X(I), 1); NEW_LINE;
    end loop;
    DO_FIND_PLATEAU_LENGTH;
  end;  -- declare block
end USE_FIND_PLATEAU_LENGTH;
```

A.4 SORT_ADAM

Here is an Adam program to sort a vector:

```
with INTEGER_TEXT_IO, TEXT_IO, PARALLEL_OUT;
use INTEGER_TEXT_IO, TEXT_IO;

procedure SORT is

-----------------------------------------------------------------------------
--   Sorts a vector into ascending order in parallel.
-----------------------------------------------------------------------------
--   The vector is distributed across an array of workers so that each worker
--   is assigned a corresponding component value.  Two diva calls are used:
--     1. A diva call counts, for each I,  how many occurences of component
--        values less than WORKER[I]'s component value are in the vector plus
--        how many occurences of component values equal to WORKER[I]'s
--        component value are to the left of or at WORKER[I].  The resulting
--        count is assigned to a local variable MY_TARGET.
--     2. MY_TARGET is the target address in the vector where WORKER[I]'s value
--        should be sent.  Another diva call carries out the communication.
-----------------------------------------------------------------------------

  N: INTEGER;

  generic
    type RANGE_TYPE is range <>;
    type DYNAMIC_VECTOR is array(RANGE_TYPE) of INTEGER;
  diva procedure COMPUTE_COUNT (A: in DYNAMIC_VECTOR;
                              GIVEN_LOC: in RANGE_TYPE;
                              GIVEN_VALUE: in INTEGER;
                              COUNT: out INTEGER) is
    -- COUNT is assigned the number of components of A that are either smaller
    -- than GIVEN_VALUE or are: equal to GIVEN_VALUE and to the left or at
    -- GIVEN_LOC.
    L, R: INTEGER;  -- results of left and right slices
  begin
    if A'LENGTH = 1 then
       if A(A'FIRST) < GIVEN_VALUE or else
          (A(A'FIRST) = GIVEN_VALUE and then A'FIRST <= GIVEN_LOC) then
          COUNT := 1;
       else
          COUNT := 0;
       end if;
    else
      COMPUTE_COUNT (A'INITIAL, GIVEN_LOC, GIVEN_VALUE, L);
      COMPUTE_COUNT (A'FINAL,   GIVEN_LOC, GIVEN_VALUE, R);
      COUNT := L + R;
    end if;
  end COMPUTE_COUNT;

  generic
    type RANGE_TYPE is range <>;
    type DYNAMIC_VECTOR is array(RANGE_TYPE) of INTEGER;
  diva procedure COMMUNICATE (A: in out DYNAMIC_VECTOR;
                             GIVEN_LOC: in RANGE_TYPE;
                             GIVEN_VALUE: in INTEGER) is
    -- If A has a subscript equal to GIVEN_LOC, then
    -- this component of A receives the value GIVEN_VALUE.
  begin
     if A'LENGTH = 1 then
        if A'FIRST = GIVEN_LOC then
           A(A'FIRST) := GIVEN_VALUE;
        end if;
     else
        COMMUNICATE (A'INITIAL, GIVEN_LOC, GIVEN_VALUE);
        COMMUNICATE (A'FINAL,   GIVEN_LOC, GIVEN_VALUE);
     end if;
  end COMMUNICATE;

begin -- SORT
  GET (N);
  PUT ("The length of the vector is ");
  PUT (N, 1); NEW_LINE;
```

```
declare
  subtype SUBRANGE is INTEGER range 1..N;
  type VECTOR is array(SUBRANGE) of INTEGER;
  X: VECTOR;

  procedure DO_SORT is

    task type WORKER_TYPE is
    private
      diva procedure MY_COMPUTE_COUNT is new COMPUTE_COUNT (SUBRANGE, VECTOR);
      diva procedure MY_COMMUNICATE is new COMMUNICATE (SUBRANGE, VECTOR);
    end WORKER_TYPE;

    WORKER: array[I: SUBRANGE] of WORKER_TYPE;

    task body WORKER_TYPE is
      COUNT: INTEGER;
      MY_TARGET: SUBRANGE;
      MY_VALUE: INTEGER;
      MY_NEW_VALUE: INTEGER;
      package PAR_OUT is new PARALLEL_OUT;
      use PAR_OUT;
    begin
      MY_VALUE := X(I);
      meet
        MY_COMPUTE_COUNT (WORKER[-].MY_VALUE, I, MY_VALUE, MY_TARGET);
        MY_COMMUNICATE (WORKER[-].MY_NEW_VALUE, MY_TARGET, MY_VALUE);
      end meet;
      X(I) := MY_NEW_VALUE;
    end WORKER_TYPE;

  begin -- DO_SORT
    null;  -- Activate the array of workers.
  end DO_SORT;

begin -- declare block
  for I in SUBRANGE loop
    GET (X(I));
  end loop;
  PUT_LINE ("The vector of values before the sort is:");
  for I in SUBRANGE loop
    PUT (X(I));
  end loop;
  NEW_LINE; NEW_LINE;
  DO_SORT;
```

```
    PUT_LINE ("The vector of values after the sort is:");
    for I in SUBRANGE loop
      PUT (X(I));
    end loop;
  end;  -- declare block
end SORT;
```

A.5 USE_PREFIX_.ADAM

The next Adam program illustrates how a parallel prefix application can be programmed:

```
with INTEGER_TEXT_IO, TEXT_IO, PARALLEL_OUT;
use INTEGER_TEXT_IO, TEXT_IO;

procedure USE_PREFIX is

  N: INTEGER;

  generic
    type RANGE_TYPE is range <>;
    type DYNAMIC_VECTOR is array(RANGE_TYPE) of INTEGER;
  diva procedure PREFIX (A: in DYNAMIC_VECTOR;
                         GIVEN_LOC: in RANGE_TYPE;
                         ANS: out INTEGER) is
    -- ANS is assigned the result of applying F to A(A'FIRST..GIVEN_LOC).
    L, R: INTEGER;
    BASE: INTEGER := 0;

    function F (X, Y: in INTEGER) return INTEGER is
    begin -- F
      return (X + Y);
    end F;

  begin -- PREFIX
     if A'LENGTH = 1 then
        if A'FIRST <= GIVEN_LOC then
           ANS := A(A'FIRST);
        else
           ANS := BASE;
        end if;
     else
        PREFIX (A'INITIAL, GIVEN_LOC, L);
        PREFIX (A'FINAL,   GIVEN_LOC, R);
        ANS := F (L, R);
    end if;
  end PREFIX;

begin -- USE_PREFIX
  GET (N);
  PUT ("The length of the vector is ");
  PUT (N, 1); NEW_LINE;
  declare
    subtype SUBRANGE is INTEGER range 1..N;
    type VECTOR is array(SUBRANGE) of INTEGER;
    X: VECTOR;
    ANSWER: INTEGER;

    procedure DO_PREFIX is

      task type WORKER_TYPE is
      private
        diva procedure MY_PREFIX is new PREFIX (SUBRANGE, VECTOR);
      end WORKER_TYPE;

      WORKER: array[I: SUBRANGE] of WORKER_TYPE;

      task body WORKER_TYPE is
        MY_ANSWER: INTEGER;
        package PAR_OUT is new PARALLEL_OUT;
        use PAR_OUT;
      begin
        meet
          MY_PREFIX (X, I, MY_ANSWER);
        end meet;
        PRINT ("The answer received by WORKER["); PRINT(I);
        PRINT ("] is "); PRINT(MY_ANSWER); PRINT_LINE;
      end WORKER_TYPE;

    begin -- DO_PREFIX
      null;  -- Activate the array of workers.
    end DO_PREFIX;

  begin -- declare block
    for I in SUBRANGE loop
      GET (X(I));
    end loop;
    PUT_LINE ("The vector is:");
    for I in SUBRANGE loop
      PUT (X(I), 1); NEW_LINE;
    end loop;
    DO_PREFIX;
  end;  -- declare block
end USE_PREFIX;
```

A.6 UPDATE

This section illustrates how a diva procedure can be used to assign values to components of two dynamic vectors. This diva procedure could be used in a program where A is the vector of health benefits paid to each worker in a company, B is the vector of total benefits paid to each worker, and FACTOR is a factor by which the health benefits will be increased. A similar diva procedure could be used to maintain some other desired relationship between the corresponding components of two or more vectors.

```
diva procedure UPDATE (A: in out DYNAMIC_VECTOR;
                       B: in out DYNAMIC_VECTOR;
                       FACTOR: in FLOAT) is
  -- Each component of A is increased by the factor FACTOR.
  -- Each component of B is increased by the amount of the increase in
  --   the corresponding component of A.
  AMOUNT_OF_INCREASE: FLOAT;
begin
  if A'LENGTH = 1 then
    AMOUNT_OF_INCREASE := A(A'FIRST) * FACTOR;
    A(A'FIRST) := A(A'FIRST) * (1.0 + FACTOR);
    B(A'FIRST) := B(A'FIRST) + AMOUNT_OF_INCREASE;
  else
    UPDATE (A'INITIAL, B'INITIAL, FACTOR);
    UPDATE (A'FINAL,   B'FINAL,   FACTOR);
  end if;
end UPDATE;
```

Appendix B

Syntax Summary

This appendix gives extended BNF for a generic diva procedure and for the meet and multiway accept constructs. The syntax for instantiations of generic diva procedures is the same as that for instantiating Ada generic procedures, except for the use of the private part of a task specification when appropriate. Other features of Adam, such as diva calls and one-dimensional indexed task declarations, are minor variations of Ada syntax, as explained earlier.

Nonterminals in the grammar with the prefix **pure_ada_code** refer to pure Ada, with static semantic restrictions as explained in Chapter 2. Each of the following nonterminals produces an identifier:

```
array_name                        discrete_range_type_name
diva_instantiation_name           dynamic_parameter_name
entry_family_name                 entry_family_index_variable
generic_diva_procedure_name       range_type_mark
simple_name                       type_mark
```

All instances of range_type_mark within the same declaration of a diva procedure must produce the same identifier. All instances of generic_diva_procedure_name within the diva procedure P must produce the identifier P.

Square brackets are not used as metasymbols, since they occur as lexical units within Adam. The metasymbols used are as follows: { } denotes zero or one instances of the item contained within, { }* denotes zero, one, or more instances of the item contained within, { }+ denotes one or more instances of the item contained within, and | denotes an alternative.

```
generic_diva_procedure_declaration
  ::=  GENERIC
         TYPE range_type_mark IS RANGE < > ;
       { TYPE type_mark IS ARRAY ( range_type_mark ) OF type_mark ; }+
       DIVA PROCEDURE generic_diva_procedure_name
         (   dynamic_parameter_declaration { ; dynamic_parameter_declaration }*
             { ; non_dynamic_declarations }  ) IS
       { pure_ada_code1 }
       BEGIN
         IF dynamic_parameter_name ' LENGTH = 1
         THEN
         { pure_ada_code2 }
         ELSE
           generic_diva_procedure_name
              ( dynamic_parameter_name ' INITIAL
                { , dynamic_parameter_name ' INITIAL }*
                { , <identifier> }* ) ;
           generic_diva_procedure_name
              ( dynamic_parameter_name ' FINAL
                { , dynamic_parameter_name ' FINAL }*
                { , <identifier> }* ) ;
         { pure_ada_code3 }
         END IF ;
       { RETURN ; }
       END   generic_diva_procedure_name  ;

dynamic_parameter_declaration
  ::=  dynamic_parameter_name { , dynamic_parameter_name }* : mode type_mark

mode
  ::=  IN   |   OUT   |   IN OUT

non_dynamic_declarations
  ::=  in_parameter_declaration
       { ; in_parameter_declaration }*                |
       out_or_in_out_parameter_declaration
       { ; out_or_in_out_parameter_declaration }*  |
       in_parameter_declaration
       { ; in_parameter_declaration }*
         ; out_or_in_out_parameter_declaration
       { ; out_or_in_out_parameter_declaration }*

in_parameter_declaration
  :: = identifier_list : IN type_mark          |
       identifier_list : IN range_type_mark
```

```
out_or_in_out_parameter_declaration
  ::=  identifier_list : OUT type_mark          |
       identifier_list : OUT range_type_mark    |
       identifier_list : IN OUT type_mark       |
       identifier_list : IN OUT range_type_mark

identifier_list
  ::=  <identifier> { , <identifier> }*

generic_diva_specification_declaration
  ::=  GENERIC
         TYPE type_mark IS RANGE < > ;
       { TYPE type_mark IS ARRAY ( range_type_mark ) OF type_mark ; }+
       DIVA PROCEDURE generic_diva_procedure_name
         (   dynamic_parameter_declaration { ; dynamic_parameter_declaration }*
             { ; non_dynamic_declarations }  ) ;

generic_diva_body_declaration
  ::=  DIVA PROCEDURE generic_diva_procedure_name
         (   dynamic_parameter_declaration { ; dynamic_parameter_declaration }*
             { ; non_dynamic_declarations }  ) IS
       { pure_ada_code1 }
       BEGIN
         IF dynamic_parameter_name ' LENGTH = 1
         THEN
         { pure_ada_code2 }
         ELSE
           generic_diva_procedure_name
              ( dynamic_parameter_name ' INITIAL
                { , dynamic_parameter_name ' INITIAL }*
                { , <identifier> }* ) ;
           generic_diva_procedure_name
              ( dynamic_parameter_name ' FINAL
                { , dynamic_parameter_name ' FINAL }*
                { , <identifier> }* ) ;
         { pure_ada_code3 }
         END IF ;
       { RETURN ; }
       END   generic_diva_procedure_name   ;

meet_statement
  ::=  MEET multiway_rendezvous_statements END MEET;

multiway_rendezvous_statements
```

```
  ::=  NULL ;  |   { diva_call_within_multiway_rendezvous }+

diva_call_within_multiway_rendezvous
  ::=  diva_instantiation_name  (  actual_parameter_list  )  ;

actual_parameter_list
  ::=  { array_name ,  |  simple_name [-] . simple_name , }+
       {  { simple_name , }*  simple_name  }

multiway_accept_statement
  ::=  ACCEPT   entry_family_name   [ entry_family_index_declaration ]
       { (  parameter_specification  { ; parameter_specification }*  )  }
       { DO  multiway_rendezvous_statements  END  { entry_family_name }  } ; |
       ACCEPT   entry_family_name   [ entry_family_index_declaration ]
       { (  parameter_specification  { ; parameter_specification }*  )  } ;

entry_family_index_declaration
  ::=  entry_family_index_variable   :   discrete_range_type_name

parameter_specification
  ::=  simple_name  { , simple_name }*  :  mode  type_mark
```