ARCTURUS USER'S GUIDE

Stephen Hunter Willson

Technical Report #241

Department of Information and Computer Science
University of California
Irvine, CA 92717

November 28, 1984

Arcturus User's Guide

Stephen Hunter Willson
Revised by Craig Snider

Programming Environment Project
Department of Information and Computer Science
University of California, Irvine

ABSTRACT

This document describes the **Arcturus** programming environment, a first realization of an advanced programming environment based on the programming language Ada. It is the first environment that supplies interactive, "statement (or declaration) at a time" execution of Ada program fragments.

Version 3.0-A

Acknowledgments: The Arcturus programming environment was designed and constructed by the Programming Environment Project at U.C. Irvine, Thomas A. Standish, Principal Investigator. Ray Klefstad, Craig Snider, Frank Tadman, Steve Whitehill, and Stephen Willson were the principal programmers on the project. Thanks to Frank Tadman and Professor Thomas A. Standish for reviewing this document. Thanks especially to Judy Bamberger and Frank Belz at TRW and Anne Brindle at Aerospace Corp. for their patient experimentation with Arcturus. Special thanks also to Richard N. Taylor, from U.C. Irvine, Anne Brindle and Larry Jansen, from Aerospace Corp., and Dave Martin from U.C. Los Angeles, for their efforts in adding tasking to Arcturus.

This work was sponsored in part by the Defense Advanced Research Projects Agency of the United States Department of Defense under contract MDA-903-82-C-0039 to the Irvine Programming Environment Project. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the United States Government.

1. Preface

Arcturus runs on Vax-family mini-computers under the Berkeley release (4.1, 4.1a, or 4.2) of the UNIX† operating system. The document, "Introduction to Arcturus", by Stephen Willson gives an executive overview of Arcturus, and should be read before wading through this document. "Introduction to Arcturus" is supplied on the magtape that contains the Arcturus files.

2. Getting started with Arcturus

The following typescript demonstrates a simple session with Arcturus.

```
% ada

Arcturus  release 3.0-A November 1, 1984

> 4+5
9
```

Typing an expression causes it to be computed and the result printed.

```
> 22*64
1408
> -5*4+2
-18
```

Next, we type in a simple function that doubles the value of its argument. Notice that if we haven't finished entering the function, the system will prompt for more input with a '+'.

† UNIX is a trademark of Bell Laboratories.

```
> function double(n : integer) return integer
+ is begin
+     return n*2;
+ end;
```

Using a procedure called 'pp', for 'pretty print', we can look at functions or procedures we have typed in or read in from a file.

```
> pp(double'sym);     -- The 'sym is required.
function double(n: in integer) return integer is
begin
   return n * 2;
end double;
```

Since functions can be part of expressions, we can just type in a call to 'double' and look at the result.

```
> double(5)
10
> double(400/2+3)
406
```

Now we'll enter a little more complicated function to compute the factorial of an integer.

```
> function fact(n : integer) return integer
+ is begin
+     if n < 2 then return 1;
+     else return n * fact (n - 1);
+     end if;
+ end;


> pp(fact'sym);
function fact(n: in integer) return integer is
begin
   if n < 2 then
      return 1;
   else
      return n * fact(n - 1);
   end if;
end fact;
```

Pretty printing it makes it more readable. We test it by getting 5! and 6!.

```
> fact(5)
120
> fact(6)
720
```

We can redefine 'fact' to 'break' when it gets to the terminating condition of the recursion. Once 'fact' has 'broken', we can look at variables and examine a 'trace' of fact's execution.

```
> function fact(n : integer) return integer
+ is begin
+   if n < 2 then
+     break;        -- this is the new line
+     return 1;
+   else return n * fact(n - 1);
+   end if;
+ end;

Warning: fact redefined in package 'usr'

> fact(3)

Break in: standard.usr.fact
```

The message indicates that 'break' was called from within 'fact'. One of the things we can do at a break is to get a backtrace of the last few statements that were executed by Arcturus. In the next two examples, we look 3 deep and 6 deep, respectively. The ampersand character ('&') character is a place holder for where the preceding entry in the backtrace belongs. The colon (':') prompt indicates that we are at a break.

```
: bktr(3);
  29.  bktr(3);
  28.  break;
  27.  if n < 2 then  & ...  else  ... end if;
```

```
: bktr(6);
  29. bktr(6);
  28. break;
  27. if n < 2 then  &  ...  else  ...  end if;
    -- In #27, '&' holds the place of 'break;'
  26. function fact(n: in integer) return integer is
begin
    &
    -- here, '&' holds the place of the if statement in #27
end fact;
  25. fact(n - 1)
  24. n * &
    -- '&' holds the place of 'fact(n - 1)'

: n   -- we can examine the value of 'n'
1     -- which is '1'
: ^D  -- Typing 'control-D' continues execution from the
      -- 'break'
6     -- and the result of fact(3) = 6
```

Arcturus allows us to enter Ada type definitions and to declare variables too. Below, we declare a type definition for an array of ten integers, then create an instance of that type with the variable 'squares'.

```
> type int_array is array(1..10) of integer;

> squares : int_array;
```

Ada statements can also be typed in. Below, we type in a for loop which fills 'squares' with the squares of the integers from one to ten.

```
> for i in 1..10 loop squares(i) := i*i; end loop;
```

We print out the value of the fifth element of the array.

```
> squares(5)
25
```

With another loop, we print out all the entries in the array. The procedure 'new_line;' prints a carriage return/line feed.

```
> for i in squares'range loop put(squares(i)); new_line; end loop;
1
4
9
16
25
36
49
64
81
100
```

The procedure 'dir;' will list out all the definitions we have entered into the environment.

```
> dir;
Visible:
Body:
   type int_array;

   squares: int_array;

   function double;
   function fact;

> ^D  -- Typing a control-D leaves the environment.
%
```

As you can see, Arcturus supports the Ada programming language in an interactive manner similar to that provided by many Lisp environments for that language. The most notable difference between Arcturus and a Lisp

environment is that Arcturus is designed to support a statically scoped, strongly typed, modular language, whereas Lisp environments support a primarily dynamic language with a relatively flat name space.

3. Details about the Arcturus Ada Interpreter

Arcturus is a fairly complicated system (a reflection of the complexities of the underlying problems of working with large systems with a strongly typed language). The remainder of this document describes the facilities available in the Ada interpreter, as well as support routines and backstage data manipulations by the interpreter that might affect your work.

3.1. Introduction to some Arcturus Terminology

This section introduces some terminology that will be used in the remaining sections of this document.

Top level: The top level of Arcturus is indicated when the system prints a '>' as the prompt. This is the level of the system you enter when it is first started. If you ever get confused, you can type a control-C to return to the top level of the system.

Break Package: The break package is indicated when the system prints a ':' as the prompt. This will happen if you try to do something illegal or if you call

the procedure 'break;'. When you are in the break package, you can type any legal Ada statement or an expression to be evaluated. To leave the break package and continue execution of your program, type control-D. To get back to the top level, type control-C. It is also possible to enter the break package by typing control-\ (that's control-backslash). If you think your program is in an infinite loop, you can type control-\ to enter the break package, examine your program's current execution environment, and then leave the break package with either control-C to return to the top level or control-D to continue execution of your program. By the way, there is special code in Arcturus to insure that control-C traps won't destroy your execution environment at a bad time. These haven't been installed for control-\

Editing characters: When you are typing to Arcturus, you can edit your input by using certain keys on the terminal. The 'delete' key will erase the last character you entered. Typing control-U will erase an entire line. Typing control-W will delete the last word.

Temporary escape: If you want to leave the environment for a short time and then return, type control-Z. This will type "stopped" and give you the Unix prompt. When you want to continue the system, type "%ada" (without the quotes) followed by a couple of carriage returns. You should see an Arcturus prompt (either '>' or ':').

Error messages: Most of the error messages you will see will be prefixed by "Error: ". This indicates that you have done something wrong. If, after care-

fully checking your program, you believe the error message to be incorrect, you should use the "bug;" procedure (described below in the section on predefined subprograms) to report the problem. Error messages prefixed by "INTERNAL ERROR: " indicate that something is wrong inside Arcturus. You will be asked to describe what you did to cause the error and then you will be put in the break package. (Likewise for "Error: " type error messages: you will end up in the break package.) Error messages prefixed by "FATAL ERROR: " indicate that Arcturus may have damaged itself beyond recovery. Arcturus tries to save all the subprograms and variables and type definitions you have entered. Re-entering Arcturus should get you running again.

The 'sym attribute: The Ada language definition does not allow procedures, functions, or packages to be passed as arguments. In general, it is difficult to *quote* a bit of Ada syntax according to the language rules. Fortunately, Ada has an escape clause which allows an implementation to define special attributes as necessary. In Arcturus, we use the implementation-defined attribute *'sym* to return the internal structure associated with an object, type definition, procedure or function name, or package name for the purposes of talking about it in the Arcturus system. The attribute 'sym returns an object of the private type known as *symbol*. Symbol is predefined in package standard. You can declare objects of type symbol, and do the usual assignment and comparison operators on them since symbol is a private type. The only way to obtain an object reference suitable for use as a *symbol* is via the attribute 'sym.

In short, whenever you call an Arcturus service that requires the name of an

object defined in Arcturus, you must quote the name with 'sym. Examples:

```
pp(fact'sym);
dir(standard'sym);
```

3.2. Reading files into Arcturus: procedures include, include_commands, and load

If you created a file "fred" with a text editor and wanted to read it into Arcturus, you would type:

```
> include("fred");
```

and this would read the text of *fred* into Arcturus, and interprets it as an Ada compilation unit, which may include only subprograms and packages right now. If include is used to read in the code:

```
use red; procedure green is ... begin ... end;
```

the use clause applies to the procedure green.

Typing:

```
> include_commands("fred");
```

reads the text of *fred* in as a sequence of commands as if they were typed at the top level of Arcturus. *fred* may contain loose expressions, declarations, and statements, in addition to those things allowed by include. If include_commands is used to read in the code:

use red; procedure green is ... begin ... end;

the use clause applies to the package usr, as if it was typed at the top level.

Typing:

> load("fred");

reads in compiled code in compilation units.

If any syntax errors were detected while reading *fred* then you would find yourself put into a text editor with the cursor flashing near the point of the error with a suitable error message displayed. (For more details on text editors in Arcturus see the section on the edit procedure and the note near the end of this document.)

If you specify a filename that is "simple" (that is, it is a Unix pathname with only one component, such as "fred", but not "./fred" or "user/fred" or "/logins/400/johnny/fred") and the file isn't found in your current directory, then Arcturus will look in a site-specific library directory for the file.

3.3. Ada Language Features Implemented in this Release

This section follows the general outline of the Ada LRM, and gives an indication of the currently implemented set of features of the Ada language available in this release of Arcturus.

2.0 Lexical Elements Complete

3.2.1 Object Declarations

The reserved word *constant* is ignored.

3.2.2 Number Declarations Not implemented.

3.3.1 The following type definitions are supported:
enumeration types
record types
access types
derived types
array types

3.3.2 The following subtype definitions are supported:
range constraint
index constraint

3.4 Derived types
Types can be derived from existing types, but no automatic inheritance of the operations available for the type occurs. One can, however, use explict type conversions to make these functions available.

3.5 Scalar types
T'FIRST and T'LAST are available.

3.5.1 Enumeration types
Character enumeration literals are not allowed.
Overloading of enumeration literals is not allowed.

3.5.2 Character types are not supported, except for the predefined character type *character*.

3.5.3 The type boolean is supported.

3.5.4 Integer types are not supported beyond the predefined type *integer* which is 32 bits wide on the VAX†.

3.5.5 Attributes available for discrete types:
'pos
'val
'succ
'pred
'image

3.5.6 Real types are not supported.

3.5.7 Floating point types are not supported beyond the type *float* which is predefined and is 32 bits wide on the VAX.

3.5.8 No attributes for floating types are defined. The operations '*', '/', '+', '-', and assignment are defined.

3.5.9 Fixed point types are not supported.

3.6 Array types are supported (both constrained and unconstrained).

3.6.1 Arcturus (erroneously) allows a declaration such as

```
s : string := "hello".
```

3.6.2 Operations on array types:
'first
'last
'range

---

† VAX is a trademark of Digital Equipment Corporation.

'length
The 'attr(n) versions are not supported.

3.6.3 The predefined type string is supported.

3.7 Record types are supported.

3.7.1 Discriminants are not supported.

3.7.3 Variant records are supported, except that the variant is never checked at runtime.

3.7.8 The attribute 'constrained is not supported.

3.8 Access types are supported.

3.9 Declarative parts follow slightly relaxed rules regarding the order of elaboration. The *export laundry* (under development) fixes this when programs are exported from Arcturus.

4.0 Names and Expressions

4.1.1 Indexed components are supported.

4.1.2 Slices are supported only in simple expressions such as

```
a(1..3) := b(2..4);
```

4.1.3 Selected component notation is supported.

4.1.4 Some attributes (as indicated) are supported in expressions.

4.2 Numeric literals are not treated as members of the type universal_integer or universal_real.

4.3 Aggregrate quantities are supported for single-dimension, positional array assignments.

4.4 Expressions
The operator *rem* is not supported.
Further, boolean operations on array types are not supported.

4.6 Type conversions for derived types and to enforce range constraints are supported. Conversion between integer and float is supported. Automatic conversion of out parameters to subprograms is not supported.

4.7 Qualified expressions are not supported.

4.8 The allocator *new* is supported.
Unchecked_deallocation is not provided. There is no garbage collector. The pragma *controlled* is not supported.

4.9 No check is made to insure that expressions which should be static are actually static. Expressions are simply evaluated at runtime.

4.10 Universal expressions are not supported.

5.0 The following statements are available:
accept
assignment
block
case
delay

entry call
exit
goto
if
loop
null
procedure call
return
select

Statements may be labeled.

6.0 Subprograms
6.2 Parameter modes *in*, *in out*, and *out* are supported.
In out and *out* parameters are passed by reference.
6.3 Subprogram bodies may have exception parts.
6.3.1 No overloading of subprogram names is permitted.
6.3.2 In-line expansion isn't supported.
6.4 Named parameters are supported.
6.4.1 Conversion of out parameters is not allowed.
6.4.2 Default parameters are supported.
6.5 Functions are supported, but no check is made to insure that parameters have mode *in* only.
6.6 No overloading is supported.
6.7 Operators may not be overloaded (or even defined as functions).

7.0 Packages are supported.
7.4 Private types are not supported.

8.0 The scoping rules are supported.
8.4 The *with* clause is not supported. The *use* clause is supported.
8.5 *Renames* is not supported.
8.6 The package Standard exists and contains a few definitions.

9.0 Tasking is supported.
9.2 Task types are not supported.
9.5 Entries, entry calls and accept statements are supported.
Entry families are not supported.
9.6 Delay statements are supported. Under BSD 4.2, resolution is 10 microseconds (for practical purposes, 10 milliseconds seems to be about the limit on a VAX 750). Under BSD 4.1, resolution is 1 second.
Type *duration* is not supported. Package *Calendar* is not implemented.
9.7 Select statements are supported.
Selective waits are supported.

Conditional entry calls are supported.
Timed entry calls are supported.
9.8 Pragma Priority is not supported.
9.9 Task and entry attributes *callable*, *terminated*, and *count* are supported.
9.10 Abort statements are not supported.
9.11 The generic library procedure *shared_variable_synchronize* is not supported

10.0 Compilation issues: n/a

11.0 Exceptions
11.1 *Constraint_error*, *numeric_error*, *storage_error*, *program_error*, and *tasking_error* are supported.
11.2 Exception handlers are supported.
11.3 The *raise* statement is supported.
11.7 It is not possible to suppress an exception check.

12.0 Generic Units are not supported.

13.0 Representation clauses are not supported.

14.0 I/O support is covered in another section.

3.4. The Morse code of comments

Comments entered into the Arcturus environment are attached to your program, and can be pretty printed with your program. Comments are hard to deal with, and we don't do a very good job with them at this time. They tend to move around a little bit and get lost. Once a comment has moved though, it will stay put. The following conventions are used to control and support the printing of comments:

-- A regular old comment
--- this comment was continued from the previous line
--. this comment is a "major" comment, and should be printed
--- on a separate line and indented the same as the current

--- indentation level of statements in the program.

Some comments are created by the system on its own! The printing of these is controlled by the "attachments" flag to the pretty printer, whereas the printing of the former is controlled by the "comments" flag to the pretty printer (see below for a description of pretty printer options).

```
--! Indicates the source of a refinement (see the document describing
--- Castor and Program Design Language)

--* Counter: nnn
-- The above comment is produced by the dynamic analysis tools
--- of Arcturus (described elsewhere).
```

3.4.1. Warning: wandering comments

Due to an implementation oversight that we haven't exactly figured out how to deal with, the mechanism which attaches comments to internal program representations is lacking finesse. The basic problem is that occasionally the parser will move a comment from where you typed it in (lexically speaking) to somewhere else. This can get to be quite annoying. The worst case is something like this:

```
--. what a great procedure
procedure yuck is
  i : integer;
begin
--. more comments ...
  null;
end;
```

transforms into

```
procedure yuck is
--. more comments
  i : integer;
begin
  null;
end;
```

Notice that the header comments are history while the comments before the first statement migrated to the top of the procedure! This horrible behavior is something you'll have to live with for now. We have a solution to this mess, but the implementation is so tedious that for some reason no one can find the time to stick it in the system. To get around the problem, put comments AFTER the syntactic unit you want them attached to. For instance,

```
procedure yuck is
--. what a great procedure (attaches to procedure header)
  i : integer;
begin
  null;
  --. more comments (attaches to null statement)
end;
```

Experiment! Once a comment has moved, it stays put.

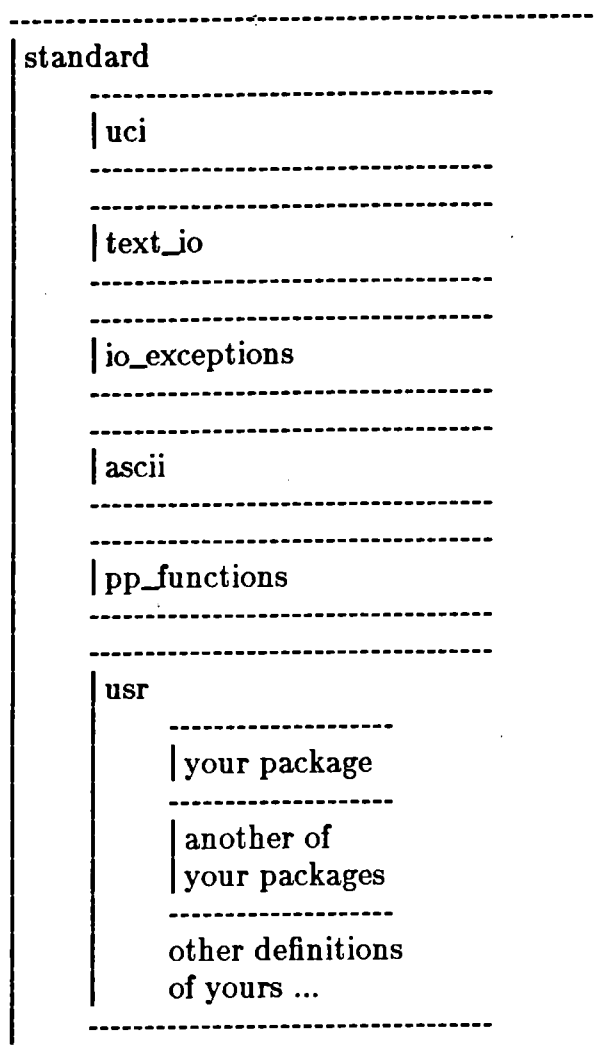
3.5. Packages available in Arcturus

The following packages are always present in the environment:

```
package standard;
-- contains definitions of integer, real, float,
-- character, string, and natural.

package io_exceptions;
-- contains the definitions for exceptions that might
-- be raised during input/output operations

package text_io;
-- contains a subset of Ada '82 input output routines.
-- See the section on input/output in Arcturus

package ascii;
-- contains definitions for non-graphic characters.
-- Ex:      "you rang the bell" & ascii.bel & "!"

package uci;
-- contains Arcturus specific procedures and functions for
-- manipulating the Arcturus environment (like bktr; break; dir;)

package pp_functions;
-- contains object and procedure definitions for use
-- with the pretty printer.

package usr;
-- contains all the definitions you enter.
```

Packages standard and usr are in an enclosing scope around your program. Packages uci, pp_functions, io_exceptions, text_io, usr, and ascii are enclosed by package standard only. The package uci is visible because it has been "used" by package standard. Package ascii must be explicitly "used" or be referenced via selected components.

The package structure looks something like this:



3.6. Some predefined subprograms

This section describes some of the commonly used subprograms available for manipulating the Arcturus environment and for interacting with the Unix environment.

3.6.1. Breaks:

These services allow you to enter the break package and to manipulate the Arcturus environment somewhat. The current implementation of the break package is such that it isn't actually a package, but merely a recursive call on the interpreter. It is dangerous, presently, to do much more than merely look around. Defining new objects (except for loop control variables) is not a good idea. But you can look around by executing Ada statements and by calling subprograms to print data structures and the like.

procedure break;

Causes your program to enter the break package. Continue with control-D. Control-C pops you back to the top-level, and kills any running tasks besides main, which is always running. You can't get back just one break level just yet.

procedure bktr(depth: in integer := integer'last);

Prints a backtrace of the statements and expressions of your program currently being evaluated by Arcturus. The default argument causes the entire backtrace to be printed.

procedure stack_trace;

Prints a backtrace of the subprogram call stack. Each subprogram currently active is printed, most recent first, followed by the name of the statically enclosing subprogram or package.

procedure where;

Prints (using the Ada selected component notation [more or less <more more than less>]) where your program is presently executing.

3.6.2. Unix services:

procedure bug;

If you find a bug in Arcturus, you can use this procedure to send a bug report to the implementors. It will prompt you for a description of your problem and then mail it to a site-specific bug report account. The Implementation Notes describe how to set the site-specific information.

procedure question;

If you have a question about Arcturus, you can use this procedure to mail a question to a site-specific user.

procedure csh(s: in string := "");

To execute a Unix command and then return to the interpreter, use csh("command");. For example, to see what users are on the system, type csh("users");. The default argument causes a child csh to be created under Arcturus which you can then use interactively. To return to Arcturus, leave the csh with control-d or "exit". Invoking a csh will read .cshrc before the shell starts accepting commands.

3.6.3. Examining the name-space:

```
procedure dir(
  symtab: in symbol := standard.usr'sym;
  filename: in string := "";
  color: in pp_color_use := no_color);
```

Dir will print a directory of the definitions in the named package. The default argument is to print a directory of the 'usr' package. (This is usually what you want since the usr package contains user-defined definitions.) Specifying a filename causes the output to be directed to that file. The color argument causes information about procedure invocation frequency to be printed on a colorscan-10 terminal. A description of the enumeration type pp_color_use is given in the section describing the pretty printer. Here is a handy redefinition you may want to try:

```
procedure dir(s : symbol := usr'sym) is
begin
   pp_functions.dir(s, filename => ",dirfile");
   csh("more ,dirfile");
end dir;
```

```
procedure edit(x: in symbol);
```

Edit will call a text editor on the named procedure or function or package. These are the only symbol types you can edit. Upon leaving the text editor, the subprogram or package will be re-read into the interpreter as if you had included it with pragma include. See the notes at the end of this document concerning the interaction of this routine with the Unix environment variable "EDITOR". Remember, to edit the function "fact", you need to type "edit(fact'sym);".

3.6.4. Random commands:

procedure quit;

Leaves the environment. Equivalent to typing control-D at the top level. See the section on the ADA.SAVE file.

procedure die;

Leaves the environment. Equivalent to typing "^Z kill %" under the C shell. Will type a message indicating memory usage, but will not update the files ada.core or ADA.SAVE. The variable uci.croak_ok must be set to true for this routine to work.

procedure fix;

This routine calls a text editor on the last command before the fix command that you entered interactively. For instance, if you typed in a big procedure and screwed it up, and got a syntax error, typing "fix;" would allow you to make a stab at correcting the trouble. If you screw up the fix command (say by typing "fx;" by mistake) then it's too late.

3.6.5. The Prettyprinter

The pretty printer takes the internal representation of an Ada program fragment and prints it according the parameters supplied. All of these parameters take default values from within the package pp_functions. If you want, you can change these values. First, here is a description of the pretty printer specification:

```
-- the pretty printer specification;
-- only the first argument is required.

procedure pp(sym: in symbol;
        filename: in string := "";
        nroff: in boolean := pp_nroff;
        vert_bars: in boolean
           := pp_vert_bars;
        comments: in boolean
           := pp_comments;
        attachments: in boolean
           := pp_attachments;
        width: in integer range 40..120
           := pp_width;
        attach_column: in
           integer range 30..110
           := pp_attach_column;
        color: in pp_color_use
           := pp_color;
        identifiers: in pp_capitalize
           := pp_identifiers;
        reserved_words: in pp_capitalize
           := pp_reserved_words;
        number: in boolean := pp_number;
        indent: in integer range 0..10
           := pp_indent;
        export: boolean := false);



    -- type definitions and default values for the pretty printer
    type pp_color_use is
       (no_color, counts, random, circular,
        depth);
    type pp_capitalize is
       (lower, upper, initial, mixed);
    pp_nroff: boolean := false;
    pp_vert_bars: boolean := false;
    pp_comments: boolean := true;
    pp_attachments: boolean := false;
    pp_width: integer range 40..120 := 70;
    pp_attach_column: integer range 30..110 := 40;
    pp_color: pp_color_use := no_color;
    pp_identifiers: pp_capitalize := lower;
    pp_reserved_words: pp_capitalize := lower;
    pp_indent: integer range 0..10 := 4;
    pp_number: boolean := false;
```

In the normal case, the pretty printer will print the given procedure, function, package, type, or object onto the terminal. If you want to change a default value for a particular parameter, you would change the corresponding object in pp_functions. For instance, to change the default width, you would set pp_width to 90. The arguments to pp are used as follows:

filename: is a string containing a Unix filename. The output will be put in this file. The default argument indicates output should go to the terminal.

nroff: if true, the output will be run through the text formatter nroff before being displayed or output to the named file. Setting this to true will cause reserved words to be underlined, the output to be paginated, and a table of contents to be created. (Great stuff!) To get this to work, you need to copy the file "rpp.titles" from wherever your Arcturus library is to your own account, and edit the file. Change the name Ada Augusta to your own name, so that your name will appear on the listings. For the nroff option to work, you need to have the tmac.X macros for nroff from UCLA installed on your Unix system.

vert_bars: if true, the output will contain vertical bars indicating the program structure.

comments: comments will be displayed if true.

attachments: if true, will cause attachments (such as PDL macros) to be displayed.

width: the maximum line length.

attach_column: Specifies the column at which all comments (both attachment comments and real comments) will displayed. Exception: Major comments (--.) ignore this argument.

color: (Note: Only useful on Datamedia Colorscan10 compatible terminals.) Given an element of the enumeration pp_color_use will have an effect as follows:

no_color	prints in black and white
counts	color depends on execution frequency
random	each construct is printed in a random color
circular	each construct is printed in a new color in a circular fashion.
depth	each construct is colored according to its syntactic depth.

identifiers: the argument specifies how identifiers should be pretty printed (all upper case, all lower case, mixed case, etc.) [Note: I think mixed case looks the best.]

reserved_words: specifies how reserved words should be printed.

number: if true, indicates that lines should be numbered.

indent: indicates how much to indent blocks.

export: since Arcturus is designed to be used to construct Ada programs that you might later want to compile, we've added this little hack to make life simpler. The problem is that we interpret a mix of Ada '80 and Ada '82. Set-

ting export => true causes the empty paren's around argumentless function calls to be removed by the pretty printer. The pretty printer will also print forward declarations for all procedures and functions in your program. This is great because Arcturus allows you to compose your program in a less structured fashion than allowed by the language definition. The export flag allows you to export code from Arcturus with a minimum of pain. This is a big hack.

3.7. Input/Output facilities

3.7.1. Terminal I/O

Output to the terminal is done with the procedure 'put'. Put usually takes one argument only, the item to be output. Examples of uses of put are:

```
put(5);            -- integer output
put("hello");  -- string output
put('c');       -- character output
put(3.4);       -- floating point output
```

Input from the terminal is done with the procedure get. The (single) argument to get must be a variable. Get will input one item of the type of its argument (either integer, float, or character).

```
declare
    i : integer;
    c : character;
    f : float;
    s : string(1..5);
begin
    get(i);          -- will input an integer
    get(f);          -- will input a floating point value
    get(c);          -- will input a character
    get(s);          -- equivalent to five getchr's.
end;
```

The procedures 'get' and 'put' are the only procedures which can be used with different typed arguments in this way. Further, get and put will only work for the types given above. If you want to print something more complicated, you must make a procedure of your own. For example:

```
declare
    type rec is record
          a,b : integer;
        end record;
    r : rec;
    procedure putrec(r : rec) is begin
          put("a => ");
          put(r.a);
          put("; b => ");
          put(r.b);
    end;
begin
    r.a := 5;
    r.b := 6;
    putrec(r);
end;
```

Note: the illusion of overloading for get and put is accomplished by special mapping of these identifiers to actual procedures named putint, putstr, putchr,

getchr, getstr, getint, etc. These procedures are stored in uci. Care must be taken if you want to define your own procedures with these names because if you define a procedure getint and then try to "get(i);" where i is an integer, your version of getint will be called if it is visible.

Further note: Due to a problem with our implementation of arrays, string arguments to the routines put, get, put_line, and get_line must be less than 255 characters in length.

3.7.2. Working with files

The following example illustrates the use of text files in Arcturus.

```
declare
   name : string := "my_file";
                    -- declare text file for use
   inp : file_type;  -- declare file descriptor: the file
                    -- descriptor is used to reference the file;
                    -- there can be more file descriptors than
                    -- files (up to about 11)

   ch : character;          -- used to store characters from the file

begin
   textedit(name);          -- call text editor to put stuff in my_file
   open(inp, in_file, name);     -- open file for reading using descriptor 'inp'
   while not end_of_file(inp) loop
         fgetchr(inp, ch);-- read a character from the file referenced
                    -- by 'inp' into 'ch'
         put(ch);   -- print on terminal
   end loop;
   close(inp);
end;
```

Notice that for terminal input/output operations 'put' and 'get' can take more than one type, but for working with files you must say which type you are going to print. 'fgetchr' means 'get a character from a file'. We didn't want to get to carried away with the subterfuge we used for overloading simple get and put. Overloading is great stuff, but a real pain in the petunia to implement. It interacts with everything.

The procedure 'Textedit' invokes a text editor on the file specified. This makes it easy to enter text into a file.

Note that the state-of-the-art in Kernel Ada Program Support Environment (KAPSE) services is presently under development nation-wide. This version of Arcturus supports only a subset of the file facilities defined for *ADA* 82.

There are several exceptions defined in the package io_exceptions that can be raised by input/output operations. See the LRM.

3.7.3. Arcturus text_io subset

Here's a list of the contents of our version of Ada '82 text_io. Presently, it's basically just a bunch of procedures that do i/o, since we don't yet do generic packages.

```
-- type file_type is limited private; --> defined in standard
--                                    for technical reasons
type file_mode is (in_file, out_file);
type count is range 0 .. integer'last;
```

```
subtype positive_count is count range 1 .. count'last;
subtype field is integer range 0..integer'last;
subtype number_base is integer range 2..16;

unbounded : constant integer := 0;

procedure create(file: in out file_type;
                 mode : in file_mode := out_file;
                 name : in string := "";
                 form : in string := "");

procedure open(file: in out file_type;
                 mode : in file_mode;
                 name : in string := "";
                 form : in string := "");

procedure close(file : in out file_type);

function standard_input return file_type;
function standard_msg return file_type;
function standard_output return file_type;
function current_input return file_type;
function current_output return file_type;
function current_msg return file_type;

procedure set_input(file: file_type);
procedure set_output(file: file_type);
procedure set_msg(file: file_type);

procedure fputreal(file: file_type; item: float; width, mantissa: integer := 0);
procedure fputstr(file: file_type; item: string);
procedure fputint(file: file_type; item: integer; width: integer := 0);
procedure fputchr(file: file_type; item: character);

-- The next four routines can be called simply as "put"
procedure putstr(item: string);
procedure putint(item: integer; width: integer := 0);
procedure putchr(item: character);
procedure putreal(item: float; width, mantissa: integer := 0);

procedure new_line(spacing: integer := 1);
procedure fnew_line(file: file_type; spacing: integer := 1);

-- The next four routines can be called simply as "get"
procedure getchr(item: out character);
procedure getreal(item: out float);
procedure getint(item: out integer);
```

```
procedure getstr(item : in out string);

procedure fgetchr(file: file_type; item: out character);
procedure fgetint(file: file_type; item: out integer);
procedure fgetreal(file: file_type; item: out float);

function end_of_file(file: file_type) return boolean;

procedure put_line(item : in string);
procedure fput_line (file : in file_type; item : in string);
procedure get_line(item : out string; last : out natural);
procedure fget_line(file : in file_type;
          item : out string;
          last : out natural);
```

Pay attention to the note above in the section describing put and get which describes the overloading that takes place when these identifier names are used as subprogram names.

Other handy input/output routines we support (which are not in text_io) are listed below.

```
procedure textedit(t: in string);
```

Calls a text editor on the file "t", allowing you to alter the object.

```
procedure type_out(t: in string);
```

Prints out on the terminal the current contents of the text object 't'. For example,

```
declare
  testing : string := "a_file";
begin
  textedit(testing);
  type_out(testing);
end;
```

Will allow you to edit the text file 'a_file' and then print its contents on the terminal.

4. Bugs, warnings, deficiencies, and stuff like that

4.1. Warning

4.1.1. It's a Prototype!

The implementation of Arcturus is under constant revision. Until otherwise noted it must be understood that the system you will be using is a **PROTOTYPE** at a rather early stage in its development. Further, the development of the system is incredibly undercapitalized given the scope of the task. As if that weren't enough, sometimes we actually have to solve novel research problems. For these reasons, we ask your cooperation and understanding as we undertake the task of making Arcturus more bullet proof and more complete.

We are interested in your responses to the system (bug reports, frustration, missing functionality, encouragement). You can send computer mail to

uci-icsc!ada on the uucp network

or

ada@uci-icsc on the InterNet

or U.S. Mail to

Arcturus
Programming Environment Project
Department of Information and Computer Science
University of California
Irvine, Ca.
92717

on matters concerning Arcturus.

4.1.2. Lack of Strict Checking:

A bothersome oversight in Arcturus is that we don't check for assignment to objects that are declared to be *constant*, including *in* parameters to subprograms. When you go to export a unit, you're likely to find that you'll get lots of error messages from the compiler. We intend to fix this sort of thing so that Arcturus Ada is completely compatible with the LRM. Actually, allowing the Arcturian style of freeform program entry to coexist with the notion of an ordered representation of a program is the topic of a thesis by Willson.

4.2. Execution Environment on Berkeley Unix 4.2

4.2.1. Using Nroff & Arcturus

If you want to use nroff to super pretty print a definition, you need to have installed on your Unix system the "X" nroff macros from UCLA.

4.2.2. Text Editors & Arcturus

Arcturus knows about a default editor to use when you use any of the procedures that invoke an editor. If you want to use a different editor than you will need to set the Unix environment variable EDITOR to the name of an editor that you wish to use. Depending on the editor named, different functionality is available. The best case is to use EMACS (by James Gosling at CMU). (Actually, it is best to use Adash, but that is another matter.) When syntax errors are detected you will be put into EMACS with two windows. One contains the offending subprogram fragment with the cursor positioned at the offending token or one past it. The other window contains a list of expected tokens. Other choices for values for the EDITOR variable are listed below.

emacs	as described above
nf	local UCI editor by Stephen Willson. Positions cursor, but only has one window. The bottom line contains an error message.
ash	template directed editor by Ray Klefstad. Same business as nf, but has special support for Ada.
vi	Berkeley screen editor. Can only position to line containing error (not the column).
none	no editor is called if an included file has a syntax error. The error message and the line and column numbers are printed (sysputf) and the error file is generated. This can be handy in Adash or when running Arcturus under EMACS. If $EDITOR is none, the edit procedure is not available.
anything else:	will print an error message, then pause 2 seconds, and execute the following "system" call: system("<EDITOR variable value> <filename>");

4.3. The User Directory & Auto-Saving of Definitions

4.3.1. General Scheme:

Arcturus will create a directory called "usr". [Note: earlier versions of Arcturus called this directory "user". You will want to rename this directory to "usr" if you have been using an earlier version of Arcturus.] This directory contains a copy of each subprogram or package you define or include in an Arcturus session. This way, if the system blows up in your face, there is a copy of the subprograms you had around when the system blew. Arcturus remembers which files you had loaded in a file called "ADA.SAVE". This file contains special pragma's which describe the subprograms and packages that you had read into Arcturus. When you restart Arcturus these files will be read in automatically when they are first referenced. In this way, if you just want to sneak into Arcturus to calculate pi or something, then you don't have to wait for these files to be read in.

The ADA.SAVE file also contains notes on which packages you had "use'd" at the toplevel. These will automatically be re-'use'-d when you reenter Arcturus. You can edit the ADA.SAVE file if you want to or if it gets damaged for some reason. The special pragma's are described below.

There is a file called "ADA.SETUP" which is also read at startup (before ADA.SAVE). You can put commands in here to execute everytime the system is started (for instance to set pretty printer defaults). My ADA.SETUP file says:

```
pp_functions.pp_width := 90;
pp_functions.pp_identifiers := pp_functions.mixed;
pp_functions.pp_reserved_words := pp_functions.initial;
pp_functions.pp_indent := 8;
uci.croak_ok := true;
uci.write_core_file := true;
```

4.3.2. The Special ADA.SAVE pragma's

There are three special pragma's that are used to get at files in the "usr" directory.

```
pragma procedure_name(fact);
pragma function_name(fib);
pragma package_name(my_pack);
```

In the examples above, "fact", "fib", and "my_pack" are the name of a procedure, function, and package, respectively, that is stored in the user directory. The use of these pragmas differs from the use of the include pragma in that the corresponding file will not be read into Arcturus until it is first referenced. This is handy if you have a lot of procedures and functions and definitions that you only use once in a while in that you don't have to pay the price of including them everytime you start up Arcturus.

Note, if you are using BSD 4.1, that since the files are stored in a Unix subdirectory, the Unix limit of fifteen character long file names applies. If you define

```
procedure this_is_my_big_procedure_1 ...
```

and then define

```
procedure this_is_my_big_procedure_2 ...
```

the first definition of big_procedure stored in the user subdirectory will get clobbered. This stuff is all temporary, until we get the session manager working (which is the subject of Steve Whitehill's thesis).

4.4. Quick startup with ada.core

4.4.1. General Scheme:

The Unix operating system doesn't provide a mechanism for saving core images of processes. As a result, it can take a noticable amount of time for Arcturus to start up while the many definitions in the standard environment are elaborated. To get around this troublesome behavior, we added two special procedures that save and restore large portions of the memory space used by Arcturus (basically from the beginning of initialized data to the program break) to and from disk files. These procedures can only be called from the top-level.

```
procedure save(filename : string := "ada.core");
-- writes a copy of the currently executing Arcturus process data
-- segments to the named file.

procedure restore(filename : string := "ada.core");
-- reads the specfied memory image into an executing Arcturus process,
-- destroying any previous definitions.
```

The core files contain a timestamp based on the creation time of the version of Arcturus that you are presently running. Arcturus will not allow an out-of-

date core file to be used to restore context information. For this reason, one should not rely exclusively on the reading and writing of core files to save definitions that have been entered during an Arcturus session, because if you move to a new version of Arcturus, the old files will be unreadable. Simply use this feature as a time saver. Remember: you can use "save" and "restore" to save and restore object and type definitions that wouldn't normally be saved in the ADA.SAVE file, but if you rely on this feature and shift to a new version of Arcturus, your old definitions will be lost.

4.4.2. Scenario for Usage:

When at the top-level of Arcturus, calling "save" will write a copy of the variable and heap storage used by Arcturus to the named file. Calling restore will read in one of these files, destroying any previous definitions. The file "ada.core" is special: when Arcturus is first invoked, it checks for this file. If it exists, then the normal startup routine of reading in library files and the contents of ADA.SAVE and ADA.SETUP is not done, and, instead, "ada.core" is read in to initialize the session.

To speed up the initialization of Arcturus, run Arcturus, and allow the normal initialization code to execute. When you get the ">" prompt, call "save;". This will write out the initialized data segment of Arcturus. Next time you invoke Arcturus, it will start up in about one or two seconds.

If you are going to give a demo, and don't want to wait for Arcturus to

start up, set up the environment the way you want it, and then call the save procedure with a suitable filename argument to save your work. When you invoke Arcturus, it will read from "ada.core", and then give you the ">" prompt. Call the restore procedure with your core file as an argument, and you'll be up and running.

```
% ada

Arcturus version ...

-- wait for Arcturus to start up (20 seconds?)
> save;            -- save initialized Arcturus
> i : integer := 3;      -- do some work ...
> save("demo.core"); -- save the current environment
> quit;
% ada             # restart Arcturus
[Context restored from 'ada.core']
> restore("demo.core");
[Context restored from 'demo.core']
> i          -- ready to roll
3
>
```

There is a variable in the *uci* package that causes the ada.core file to be written automatically when the system is existed via ^D or the *quit* procedure. Setting "uci.write_core_file := true;" will turn on this mechanism. In this way, you get quick start up of Arcturus without having to worry that your ada.core and ADA.SAVE files will get out of sync. Then, if you have a new version of Arcturus installed, you can recreate the bulk of your execution environment from the ADA.SAVE file. Of course, any object definitions not enclosed in packages will be lost. For more information on this sort of stuff, see "How to Export Files from Arcturus", by Stephen Willson, and distributed with this tape.

"How can anyone feel so important when we know that death is stalking us?"

-- don Juan in *Journey to Ixtlan*
by Carlos Castaneda

The End