Ada/CS

An Instructional Subset of the Programming Language Ada

James E. Archer, Jr.

TR79-395

Revised November 1979

Department of Computer Science
Cornell University
Ithaca, New York 14853

This work was supported in part by a grant from the NSF MCS77-08198.
Ada/CS Language Subset Definition

6.2
subprogram_specification ::= subprogram_nature_designator [formal_part] [return_type_mark]

subprogram_nature ::= function | procedure

designator ::= identifier

formal_part ::= (parameter_declaration [, parameter_declaration])

parameter_declaration ::= identifier_list : mode type_mark [:= expression]

mode ::= [in] | out | in out

6.4
subprogram_body ::= subprogram_specification is declarative_part
begin
  sequence_of_statements
end identifier;

7.1
module_specification ::= module_nature_identifier [is declarative_part] end Identifier]

module_nature ::= package

module_body ::= module_nature body_identifier is declarative_part
end identifier;

8.3
visibility_restriction ::= restricted [visibility_list]

visibility_list ::= (unit_name [, unit_name])

8.4
use_clause ::= use module_name [, module_name]

4.1 Ada/CS Keywords

The following words are reserved in Ada and Ada/CS.

all
else
function
mod
out
return
when
and
elsif
goto
not
package
reverse
while
assert
end
if
null
procedure
separate
xor
begin
exception
in
of
range
then
begin
exit
is
null
or
record
then
constant
for
loop
others
restricted
use
Ada/CS Language Subset Definition

The following names are predefined in Ada/CS, and are treated as reserved words by the implementation.

- boolean
- character
- delete
- false
- first
- float
- get
- integer
- last
- length
- max_int
- min_int
- ord
- pred
- put
- rep
- str ing
- succ
- true
- val

The following words are Ada reserved words, but correspond to features not implemented in Ada/CS. To maintain compatibility, these words are prohibited in Ada/CS programs.

- abort
- access
- at
- delay
- delta
- digits
- do
- entry
- generic
- initiate
- new
- packing
- private
- raise
- renames
- subtype
- task

Acknowledgments

Richard Conway has been the guiding force behind this work. Andrew Shore and Tim Teitelbaum have provided helpful comments on the progress of the subset. John Goodenough and David Gries made helpful suggestions on the previous version of the subset.

References

4.5

logical_operator ::= and | or | xor
relational_operator ::= = | /= | < | <= | > | >=
adding_operator ::= + | - | &
unary_operator ::= + | - | not
multiplying_operator ::= * | / | mod

4.6

qualified_expression ::= type_mark(expression)

5

sequence_of_statements ::= [statement]

statement ::= simple_statement | compound_statement
            | <<identifier>> statement

simple_statement ::= assignment_statement | exit_statement
                   | goto_statement
                   | null;

compound_statement ::= if_statement | case_statement | loop_statement

5.1

assignment_statement ::= variable := expression;

5.2

subprogram_call_statement ::= subprogram_call;

subprogram_call ::= subprogram_name[(parameter_association [, parameter_association])]

parameter_association ::= actual_parameter

actual_parameter ::= expression

5.3

return_statement ::= return [expression];
5.4

if_statement ::= 
   if condition then
   sequence_of_statements
   [elsif condition then
   sequence_of_statements]
   [else
   sequence_of_statements]
   end if;

condition ::= expression [and then expression]
           | expression [or else expression]

5.5

case_statement ::= 
   case expression of
   [when choice [; choice] => sequence_of_statements]
   end case

5.6

loop_statement ::= [iteration_specification] basic_loop

basic_loop ::= 
   Loop
   sequence_of_statements
   end loop [Identifier]

iteration_specification ::= 
   for loop_parameter in [reverse] discrete_range
   | while condition

loop_parameter ::= identifier

5.7

exit_statement ::= exit [identifier] [when condition];

5.8

goto_statement ::= goto identifier;

5.9

assert_statement ::= assert condition;

6.1

declarative_part ::= [use_clause] [declaration] {body}
body ::= {visibility restriction} unit_body
unit_body ::= subprogram_body | module_specification | module_body
2.4
number ::= integer_number | approximate_number
integer ::= digit {digit}
approximate_number ::= integer.integer [E exponent] | integer E exponent
exponent ::= [+] integer | − integer
2.5
character_string ::= "{characters}"
3.1
declaration ::= object_declaration | type_declaration
3.2
object_declaration ::= [constant] type [:= expression];
identifier_list ::= identifier [, identifier]
3.3
type ::= type_mark
type_definition ::= enumeration_type_definition | integer_type_definition
array_type_definition | record_type_definition
type_mark ::= type_name
constraint ::= range_constraint
type_declaration ::= type identifier is type_definition;
3.5
range_constraint ::= range range
range ::= simple_expression .. simple_expression
3.5.1
enumeration_type_definition ::= (enumeration_literal, {enumeration_literal})
enumeration_literal ::= identifier | character_literal
3.5.4
integer_type_definition ::= range_constraint
3.6
array_type_definition ::= array (index [, index]) of type_mark;
index ::= discrete_range | type_mark
discrete_range ::= [type_mark range] range
3.6.2
aggregate ::= (component_association [, component_association])
component_association ::= [choice [; choice] => ] expression
choice ::= simple_expression | discrete_range | others

3.7
record_type_definition ::= record
     component_list
     end record

component_list ::= {object_declaration}

4.1
name ::= identifier     | indexed_component
     | selected_component | predefined_attribute
indexed_component ::= name(expression [, expression])
selected_component ::= name . identifier
predefined_attribute ::= name ' identifier

4.2
literal ::= number | enumeration_literal | character_string

4.3
variable ::= name [(discrete_range)] | name.all

4.4
expression ::= relation [and relation]
     | relation [or relation]
     | relation [xor relation]
relation ::= simple_expression [relational_operator simple_expression]
     | simple_expression [not] in range
     | simple_expression [not] in type_mark [constraint]
simple_expression ::= [unary_operator] term [adding_operator term]
term ::= factor [multiplying_operator factor]
factor ::= primary
primary ::= literal | variable | subprogram_call | aggregate
     | qualified_expression | (expression)
3.1 Ada Features Omitted in the CS Subset

Ada/CS has been defined by removing constructs from the full language and limiting some that remain. The major constructs that have been removed include:

1. Facilities for definition of tasks and all concommitent concurrent programming features;
2. User-defined generic procedures;
3. Name overloading or redefinition.
4. Control over variable or record representation, variant records, access variables.
5. The exception mechanism.

3.2 Further Restrictions Imposed in Ada/CS

The principal semantic and syntactic restrictions attached to the constructs included in Ada/CS are the following:

expressions

1. Array or slice expressions are not allowed.
2. Exponentiation is not included in the subset [6].

subprograms

1. Actual parameters corresponding to out and in out formal parameters must represent locations which can legally be assigned values of the appropriate type.
2. Formal parameter array subscript range values cannot be specified.
3. Only function subprograms can return values; procedure side-effects are not allowed in expressions.
4. Keyword parameter mechanisms are not allowed.
5. Subprograms cannot be used to redefine operators.

declarations

1. There are no anonymous types.
2. No overloading of names is allowed.

case statements

1. At least two choices must be specified.
2. A "when others" clause is required if all legal cases are not syntactically guaranteed. The reserved value "uninitialized" is not part of others.

goto statements

1. Goto statements may transfer control forward only.
2. A target statement may only be contained within compound statement if all of the gotos referencing it are also contained within the same clause of the compound statement.
3.3 Modifications to the STANDARD package

In order to provide a suitable environment for the beginning programmer, Ada/CS implementations will require a significantly modified STANDARD package, including:

1. The addition of simple file-processing procedures (similar to PL/I GET/PUT LIST or PASCAL READ/WRITE) along with sufficient status monitoring functions to replace exception processing [2].
2) Limit basic types to BOOLEAN, CHARACTER, FLOAT, INTEGER, and STRING.
3. Define types for one- and two-dimensional arrays of BOOLEAN, FLOAT and INTEGER, named type_VECTOR and type_MATRIX, for example:

   type FLOAT_VECTOR is array (INTEGER) of FLOAT;

3.4 Visibility of Names in Ada/CS

Ada provides a variety of mechanisms by which the programmer can control the visibility of names and access to separately compiled program units. For Ada/CS, these visibility "restriction" mechanisms are not useful. Unfortunately, Ada essentially requires that these mechanisms be available and that at least some of them be used. The choice of facilities is sufficiently complicated as to confuse experienced programmers, much less students. Hoare suggests that separate and restricted be omitted in favor of the pragmas ENVIRONMENT and INCLUDE [6].

Ideally, Ada/CS would have no explicit mechanism for providing the inter-program name correspondence; the implementation would be responsible for the location and correctness of separate program units. Any such automatic system violates (by omission) Ada syntax. As a compromise, the Ada/CS language adheres to standard Ada syntax. The Cornell Ada/CS implementation emphasizes error-correction in all phases of program development. In this spirit, the implementation will provide the visibility "restriction" statements or pragmas necessary to meet user needs; the user will not need to enter them.

4. Ada/CS Syntax

The syntax summary given is a proper subset of the Ada syntax presented in [1]. All productions are created by removing phrases from the reference version. The section numbers from the original have been preserved.

2.3

identifier ::= letter ([underscore] letter_or_digit)

letter_or_digit ::= letter | digit

letter ::= upper_case_letter
From the beginning of the project that lead to the promulgation of Ada, there have been clear injunctions against any definition of subsets of the complete language. Presumably the purpose was to encourage standardization, but perhaps also to pressure the designers to be frugal in their proposals by denying them the escape that certain exotic features would be ignored in practical subsets. However, since the initial design phase is now more or less completed, that second objective is no longer relevant, and judging from the language chosen, may not have been entirely effective, in every application, and if there is a well-defined class of applications

Whatever its intent, the subset prohibition is a curious and unworkable prohibition at best. Presumably, individual programmers are not required to use the full language that do not need certain language features, then the instructive value of carefully specifying an appropriate subset should not be ignored.

There are also numerous and significant advantages to having compilers that support a formal subset. There are obvious economies: subset compilers are less expensive to develop, faster in operation, and executable on smaller systems. In addition, a compiler can provide more effective diagnostics if it knows that certain language features will not be used. (The PL/I experience has been very clear in this regard.)

It seems certain that if Ada becomes a viable language, it will be subjected to subsetting. The only real issue is whether there will be a plethora of ad hoc, mutually incompatible subsets, or a small number of carefully designed standard subsets. Our preference is for the latter, and we propose a subset for one large and potentially important class of application.

This report is a revision of a previously proposed subset. Since the original, we have talked with many persons active in the Ada definition effort and received their comments on the original proposal. In addition, many of Hoare's [6,7] suggestions have been incorporated. The principal modifications regard:

1. Packages. Packages were not a part of the original subset proposal; they have been added, with some restrictions.

2. Visibility of names. The original subset proposal modified the rules for separately compiled procedures so that names were unique, but no explicit mechanism was used for locating them. Goodenough [5] pointed out that this made Ada/CS a dialect rather than a subset. The Cornell Ada/CS system will provide the required Ada constructs to effect linkage.

3. Exceptions. The single exception from the previous subset, END_OF_FILE, has been removed.
Ada/CS Language Subset Definition

1. An Instructional Subset

The target application for this subset proposal is introductory programming instruction, as provided to general university audiences by computer science faculty. Note that the intent is to serve a much larger population than just "computer science majors". FORTRAN is most commonly used for this purpose at present, with PL/I, PASCAL, ALGOL-W and BASIC contending for a distant second place. The choice of language for this purpose is a highly political question, and it seems safe to say that FORTRAN would not be the preference of the majority of instructors.

There are two models for this subset. One is PASCAL. Except for a few quirks (notably non-dynamic array parameters, and limited string processing) PASCAL is well-suited to this purpose, and its rapid recent growth attests to the recognition of that fact. The second is PL/CS — a very carefully chosen, highly-disciplined subset of PL/I [4]. PL/CS was designed to strongly encourage the contemporary view of preferred programming practice, and at the same time facilitate the formal verification of the correctness of programs. It is interesting that most of the PL/CS restrictions anticipated those adopted in Ada — for example, explicit preatory declarations, elimination of function side-effects, restrictions on access to control variables in the body of an indexed loop.

Whatever its merits for systems programming for embedded systems, the Ada designers have produced a language that encloses a very attractive instructional language. Although it would be too much to expect the PASCAL purists to admit it, an Ada subset is significantly better than PASCAL for this purpose, and PASCAL certainly dominates the other incumbents in this regard.

A richer Ada subset would be desirable for more advanced instructional purposes. Hoare has suggested the inclusion of variant records and rudimentary tasking in the current subset [7]. While these features would be useful in many instructional contexts, they should be the province of a larger, intermediate subset that still does not include all of Ada.

2. An Implementation of Ada/CS

An implementation of Ada/CS is currently under way at Cornell. This is in the form of a program development system that provides an integrated file system, editor, translator and execution supervisor [3]. The system will operate either on a dedicated microprocessor or a large-scale time-shared computer. A similar system for the PL/CS language is already in classroom use [8].

3. Summary of Ada/CS

The fastest way to describe this subset to someone generally familiar with Ada is to enumerate the omissions and restrictions relative to the full language. This is done in the following lists.