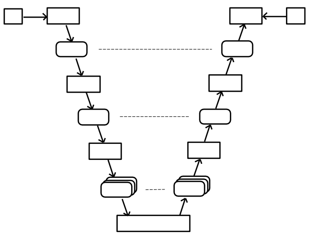
National Technical University of Athens
School of Electrical Engineering & Computer Engineering
Department of Information Technology & Computer
Engineering Software Technology Laboratory

Compilers 2023 Working
topic

The Grace language

Grace Murray Hopper (1906-1992), Rear Admiral, U.S. Navy



Compilers

`http://courses.softlab.ntua.gr/compilers/`

Teacher: *Kostis Sagonas*

Athens, February 2023

SUBJECT:

Have each group of at most two students design and implement a compiler for the Grace language. The implementation language can be one of C/C++, Java, SML, OCaml, Haskell or Python, but keep in mind that parts of the compiler and tools that may be useful may not be available in some of the above languages, in which case you will have to implement them yourself. The choice of another implementation language can be made in consultation with the instructor. It is allowed and also recommended to use tools, e.g. `flex/MLLex/ocamllexAlex`, `bison/MLYacc/ocamlyaccHappy`, `JavaCC`, etc., as well as `LLVM`. More information related to these tools will be given in the deliverables.

Deliverables, dates AND grading

The parts of the compiler and the module distribution are shown in the table below. The delivery dates are listed on the course page.

Section of the compiler	*Units*	*Bonus*
Verbal analyser	0.5	-
Editorial analyst	0.5	-
Semantic analyser	1.0	0.5
Intermediate code	1.0	-
Optimization	1.0	0.8
Final code	1.0	0.7
Overall work and report	5.0	2.0

For the different parts of the work, each team must deliver the corresponding code in electronic form and clear instructions for the production of an executable program demonstrating the operation of the respective part from this code. Late assignments will be given a lower mark in inverse proportion to the time taken. Please **do not hand in printed assignments**! The format and contents of deliverables, including the final report, must conform to the guidelines given in Section 4 herein.

Optional sections AND bonus units

The total number of units in this paper is 7. Of these, 2 credits are bonus (which means that the total number of credits in the course is 12) and correspond to the following sections of the paper, which are optional:

- (50%)Intermediate code optimization, with data flow and control analysis.
- (50%) Register binding and optimization of final code.

Contents

1 Description of the language Grace

Grace is a simple imperative programming language. (Its aesthetics are forgivable if one considers its design.) Its main features in brief are as follows:

- Simple structure and syntax of commands and expressions similar to Pascal and C.
- Basic data types for characters, integers and tables.
- Simple functions, pass by value or pass by reference.
- Scope of variables as in Pascal.
- Library of functions.

More details of the language are given in the following paragraphs.

1.1 Verbal units

Grace language lexical units are divided into the following categories:

- The *keywords*, which are the following:

```
and       char      div       do          else      fun       if
int       mod       not       nothing     or        ref       return
then      var       while
```

- *Names*, which consist of a letter of the Latin alphabet, possibly followed by a series of letters, decimal digits or underscore characters. Names must not coincide with the keywords mentioned above. Lower and upper case letters are considered different.
- Unsigned *integer constants* consisting of one or more decimal places. Examples of integer constants are the following:

```
0      42      1284      00200
```

- *Fixed characters*, consisting of a character within single quotation marks. This character may be any common character or escape sequence. Common characters are all printable characters except single and double quotation marks and the \ (backslash) character. Escape sequences start with the character
\ (backslash) and are described in Table 1. Examples of fixed characters are
following:

```
'a'      '1'      '\n'      '\''      '\x1d'
```

- *Fixed strings*, consisting of a sequence of common characters or escape sequences within double quotation marks. Strings must begin and end on the same program line. Examples of fixed strings are the following:

```
"abc"            "Route66"            "Helloworld!\n"
      "Name:\t\"DouglasAdams\"\nValue:\t42\n"
```

- The *symbolic operators*, which are the following:

```
+      -      *      =      #      <      >      <=      >=
```

Table 1: Escape sequences.

Sequence escape	Description
`\n`	the line feed character
`\t`	the tab character
`\r`	the carriage return character at the beginning of the line (carriage return)
`\0`	the character with ASCII code 0
`\\`	the character \ (backslash)
`\'`	the character `'` (single quotation mark)
`\"`	the character `"` (double quotation mark)
\xnn	the character with ASCII code *nn* in the hexadecimal system

- The *separators*, which are the following:

```
(    )    [    ]    {    }    ,    ;    :    <-
```

In addition to the verbal units mentioned above, a Grace program may also contain the following, which separate verbal units and are ignored:

- *Blank characters*, i.e. sequences consisting of space, tab characters, line feed characters or carriage return characters.
- *(Possible) multi-line comments*, which begin and end with the `$$` character sequence. Comments of this form may not be nested.
- *Single-line comments*, which begin with the `$` character, are not followed by another `$` (because then it is assumed that a comment of possibly several lines begins), and end with the end of the current line.

1.2 Data TYPES

Grace supports two basic data types:

- `int`: integers of size at least 16 bits (-32768 to 32767), and
- `char`: characters.

In addition to the basic formulas, Grace also supports matrix formulas, denoted by *t*`[`*n*`]`. The *n* must be an integer constant with a positive value and *t must* be a valid formula. It also indirectly supports a logical expression type, but this is used only in the conditions of `if` and `while` statements. This type should not be confused with the `int` and `char` data types.

1.3 Structure of the programme

Grace is a block structured language. A program has roughly the same structure as a Pascal program. Building blocks can be nested within each other and the scoping rules are the same as in Pascal. The main program is a building block that returns no result and takes no parameters.

Each module may optionally contain:

- Variable declarations.

- Subprogram definitions.
- Subprogram declarations, the definitions of which will follow in the same scope.

1.3.1 Variables

Variable declarations are made with the `var` keyword. This is followed by one or more variable names, the delimiter `:`, a data type and the delimiter `;`. More consecutive variable declarations can be made by repeating the `var` keyword. Examples of declarations are:

```
var i       int;
var x, y, z : int;
var s       : char[80];
```

1.3.2 Building units

A module is *defined* by writing its header, local statements and body. The header starts with the keyword `fun`. It contains the name of the module, its formal parameters (optional) and the return type. The return type of modules that do not return a result (cf. procedures in Pascal) is `nothing`. The return type cannot be a table type. In the following we will call *procedures* the modules with return type `nothing` and *functions* the remaining modules.

The syntax of the standard parameters is similar to that of Pascal. Each standard parameter is identified by its name, its type and the way it is passed. The Grace language supports passing parameters by value and by reference. If the declaration starts with the keyword `ref`, then the declared parameters are passed by reference, otherwise they are passed by value. Parameters of the table type shall be passed by reference. In table types used for standard parameter declarations, the size of the first dimension may be omitted.

The following are examples of module definition headings.

```
fun p1 () : nothing
fun p2 (n : int) : nothing
fun p3 (a, b : int; ref c : char) : nothing
fun f1 (x : int) : int
fun f2 (ref s : char[]) : int
fun matrix_mult (ref a, b, c : int[10][10]) : nothing
```

The local statements of a module follow the header. Grace follows Pascal's scoping rules regarding the visibility of variable names, building unit names, and parameter names.

In the case of mutually recursive building blocks, the name of a building block needs to appear before its definition. In this case, in order not to violate the scope rules, the header of this building unit (without its local statements or body) must be preceded by a *declaration of* the header of this building unit (without its local statements or body) terminated by the delimiter `;`.

1.4 Expressions AND conditions

Each Grace expression has a unique type and can be evaluated by giving a value of that type as a result. Expressions fall into two categories: those that yield lvalues, described in Section 1.4.1, and those that yield rvalues, described in Sections 1.4.2 to 1.4.4. These two types of values are named after their position in an assignment statement: an lvalue appears in the left-hand member of the assignment, while an rvalue appears in the right-hand member.

Grace conditions are described in Section 1.4.3. They are used only in conjunction with `if` and `while`, and their computation results in a logical value (true or false).

Expressions and conditions may appear in parentheses, used for grouping purposes.

1.4.1 Lvalue

Lvalues represent objects that take up space in computer memory during program execution and which can contain values. Such objects are variables, module parameters, table elements and fixed strings. Specifically:

- The name of a variable or parameter is lvalue and corresponds to the object in question. The type of lvalue is the type of the corresponding object.
- Fixed strings, as described in Section 1.1, are lvalue. They are of type `char` `[`*n*`]`, where *n* is the number of characters contained in the string plus one. Each such lvalue corresponds to an object of table type, in which the characters of the string are stored in order. At the end of the table, the character `'\0'` is automatically stored, according to the C convention for strings. Fixed strings are the only type of constant table type allowed in Grace.
- If *l* is an lvalue of type *t*`[`*m*`]` and *e* is an expression of type `int`, then *l*`[`*e*`]` is an lvalue of type *t*. If the value of the expression *e* is the non-negative integer *n*, then this lvalue corresponds to the element with index *n of* the table corresponding to *l*. The numbering of the elements of the table starts from zero. The value of *n* must not exceed the actual limits of the matrix ($n < m$).

If an lvalue is used as an expression, the value of that expression is equal to the value contained in the object corresponding to the lvalue.

1.4.2 Fixed

Grace language rvalues include the following constants:

- Unsigned integer constants, as described in Section 1.1. They have type `int` and their value is equal to the non-negative integer they represent.
- Fixed characters, as described in Section 1.1. They have `char` type and their value is equal to the character they represent.

1.4.3 Operators

Grace's operators are divided into operators with one or two operands. Operators with one operand are written before the operand (prefix), while operators with two operands are always written between the operands (infix). The evaluation of the endings is done from left to right. Operators with two endings necessarily evaluate both endings, with the exception of the `and` and `or` operators, as described below.

All Grace operators result in rvalue or condition. They are described in detail below.

- Operators with a `+` and `-` operator implement the prefix operators. The operand must be an expression of type `int` and the result is an rvalue of the same type.

Table 2: Priority and attractiveness of Grace's operators.

Operators	Description	Number of finished products	Location and convenienc e
`+ -`	Signs	1	prefix
`* div mod`	Multiplying operators	2	infix, left
`+ -`	Additive operators	2	infix, left
`= # > < <= >=`	Comparison operators	2	infix, none
`not`	Reasonable denial	1	prefix
`and`	Logical coupling	2	infix, left
`or`	Logical decoupling	2	infix, left

- The operator with a `not` operator implements the logical negation. Its operand must be a condition and so must its result.
- Operators with two operands `+`, `-`, `*`, `div` and `mod` implement arithmetic operations. The operands must be expressions of type `int` and the result is rvalue of the same type.
- The operators `=`, `#`, <, >, `<=` and `>=` implement the comparison relations between numbers. The operands must be expressions of the same type `int` or `char` and the result is a condition.
- The `and` and `or` operators implement the logical operations of conjunction and disjunction respectively. The operands must be conditions and so must the result. The evaluation of conditions using these operators is done by *shortcircuiting*. That is, if the result of the condition is known from the evaluation of the first operand alone, the second operand is not evaluated at all.

Table 2 defines the priority and attractiveness of Grace's operators. The rows higher in the table contain operators of higher priority. Operators in the same row have the same priority.

1.4.4 Function call

If f is the name of a function with return type t (not `nothing`), then the expression $f(e_1, \dots, e_n)$ is a rvalue with type t. The number of real parameters n must coincide with the number of formal parameters of f. Also, the type and kind of each real parameter must coincide with the type and passing mode of the corresponding formal parameter, according to the following rules.

- If the standard parameter is of type t and is passed by value, then the corresponding actual parameter must be an expression of type t.
- If the standard parameter is of type t and is passed by reference, then the corresponding actual parameter must be a lvalue of type t.

When calling a module, the actual parameters are evaluated from left to right.

1.5 Commandments

The commands supported by the Grace language are the following:

- The empty command `;` which does not take any action.
- The assignment command `ℓ <- e;` which assigns the value of the expression *e* to the lvalue ℓ. The lvalue ℓ must be of type *t* which is not an array type, while the expression *e* must be of the same type *t*.
- The compound instruction, consisting of a series of valid instructions between hooks `{` and `}`. These commands are executed sequentially.
- The procedure call command f (e_1 , . . . , e_n) ; under the same conditions as in Section 1.4.4, except that the return type of *f* must be `nothing`.
- The control command `if` *c* `then` s_1 `else` s_2 , where *c* must be a valid condition and s_1 , s_2 must be valid commands. The `else` part is optional. The semantics of this command is as in C.
- The control command `while` *c* `do` *s*, where *c must be* a valid condition and *s* a valid command. The semantics of this command is as in C.
- The jump instruction `return` *e* `;` which terminates the execution of the current module and returns the value of *e as the* result. If the current building block has a return type of `nothing` then the *e* expression shall be omitted. Otherwise, the expression *e* must have the same return type as the return type of the current building block.

1.6 Runtime library

Grace supports a set of predefined building blocks, implemented in x86 assembly as a runtime library. They are visible in each module unless they are obscured by variables, parameters, or other modules with the same name. Their declarations are given below and their function is explained.

1.6.1 Entrance AND exit

```
fun writeInteger (n : int) : nothing; fun
writeChar  ( c  :  char)  :  nothing;  fun
writeString (ref s : char[]) : nothing;
```

These building blocks are used to print values belonging to Grace's basic types, as well as to print strings.

```
fun readInteger ()                          : int;
fun readChar    ()                          : char;
fun readString (n : int; ref s : char[]) : nothing;
```

Similarly, the above building blocks are used to insert values belonging to Grace's basic types and to insert strings. The `readString` procedure is used to read a string up to the next line change character. Its parameters specify the maximum number of characters (including the final `'\0'`) allowed to be read and the character table in which they are to be placed. The line break character is not stored. If the table size is exhausted before a line break character is encountered, reading shall be resumed later from the point where it was interrupted.

1.6.2 Conversion functions

```
fun ascii (c : char) : int;
fun chr   (n: int) : char;
```

The first returns the ASCII code of a character. The second converts a value of type `int`, which must contain the ASCII code of a character, to that character.

1.6.3 String management

```
fun strlen (ref s : char[])        int;
fun strcmp (ref s1, s2 : char[])   : int;
fun strcpy (ref trg, src : char[]) : nothing;
fun strcat (ref trg, src : char[]) : nothing;
```

These modules have exactly the same function as their synonyms in the C function library.

2 Grace's Complete Grammar

The syntax of the Grace language is given below in EBNF format. The grammar below is *ambiguous*, but most ambiguities can be overcome if one takes into account the priority and applicability rules for operators, as described in Table 2. The symbols ⟨id⟩, ⟨intconst⟩, ⟨*charconst*⟩ and ⟨*stringliteral*⟩ are terminal symbols of the grammar.

⟨*program*⟩ ::= ⟨*funcdef*⟩

⟨*funcdef*⟩ ::= ⟨*header*⟩ (⟨*localdef*⟩)* ⟨*block*⟩

⟨*header*⟩ ::= "fun" ⟨*id*⟩ "(" [⟨*fpardef*⟩ (";" ⟨*fpardef*⟩)*] ")" ":" ⟨*rettype*⟩

⟨*fpardef*⟩ ::= ["**ref**"] ⟨*id*⟩ ("," ⟨*id*⟩)* ":" ⟨*fpartype*⟩

⟨*datatype*⟩ ::= "**int**" | "**char**"

⟨*type*⟩ ::= ⟨*datatype*⟩ ("[" ⟨*intconst*⟩ "]")*

⟨*rettype*⟩ ::= ⟨*datatype*⟩ | "**nothing**"

⟨*fpartype*⟩ ::= ⟨*datatype*⟩ ["[" "]"] ("[" ⟨*intconst*⟩ "]")*

⟨*localdef*⟩ ::= ⟨*funcdef*⟩ | ⟨*funcdecl*⟩ | ⟨*vardef*⟩

⟨*funcdecl*⟩ ::= ⟨*header*⟩ ";"

⟨*vardef*⟩ ::= "var" ⟨*id*⟩ ("," ⟨*id*⟩)* ":" ⟨*type*⟩ ";"

⟨*stmt*⟩ ::= ";" | ⟨*lvalue*⟩ "<-" ⟨*expr*⟩ ";" | ⟨*block*⟩ | ⟨*funccall*⟩ ";"
    | "if" ⟨*cond*⟩ "then" ⟨*stmt*⟩ ["else" ⟨*stmt*⟩]
    | "while" ⟨*cond*⟩ "do" ⟨*stmt*⟩ | "**return**" [⟨*expr*⟩] ";"

⟨*block*⟩ ::= "**{**" (⟨*stmt*⟩)* "**}**"

⟨*funccall*⟩ ::= ⟨*id*⟩ "(" [⟨*expr*⟩ ("," ⟨*expr*⟩)*] ")"

⟨*lvalue*⟩ ::= ⟨*id*⟩ | ⟨*stringliteral*⟩ | ⟨*lvalue*⟩ "[" ⟨*expr*⟩ "]"

⟨*expr*⟩ ::= ⟨*intconst*⟩ | ⟨*charconst*⟩ | ⟨*lvalue*⟩ | "(" ⟨*expr*⟩ ")" | ⟨*funccall*⟩
    | ("+" | "-") ⟨*expr*⟩ | ⟨*expr*⟩ ("+" | "-" | "*" | "**div**" | "**mod**") ⟨*expr*⟩

⟨*cond*⟩ ::= "(" ⟨*cond*⟩ ")" | "not" ⟨*cond*⟩ | ⟨*cond*⟩ ("**and**" | "**or**") ⟨*cond*⟩
    |⟨*expr*⟩ ("=" | "#" | "<" | ">" | "<=" | ">=") ⟨*expr*⟩

3 Examples

This section gives five examples of Grace language programs, the complexity of which varies considerably. For some of these examples (or similar examples), you can find the initial, intermediate (unoptimized) code, the form of the module activation records, and the final code in corresponding language description booklets given as assignments in the same course in previous years, via the course website.

3.1 Say HELLO!

The following example is one of the simplest programs in the Grace language that produces a result visible to the user. This program simply prints a message.

```
fun hello () : nothing
{
   writeString("Hello world!\n");;
}
```

3.2 THE TOWERS of Hanoi

The following program solves the Hanoi Towers problem. A brief description of the problem is given below.

There are three pillars, on the first of which are passed *n* the multitude of rings. The outer diameters of the rings are different and they are passed from bottom to top in descending order of outer diameter, as shown in Figure 1:

- Only one ring is allowed to be transferred at a time, from one pole to another pole.
- It is forbidden to place a ring with a larger diameter over a ring with a smaller diameter.

The Grace language program that solves this problem is given at the beginning of the next page. The `hanoi` procedure is recursive.

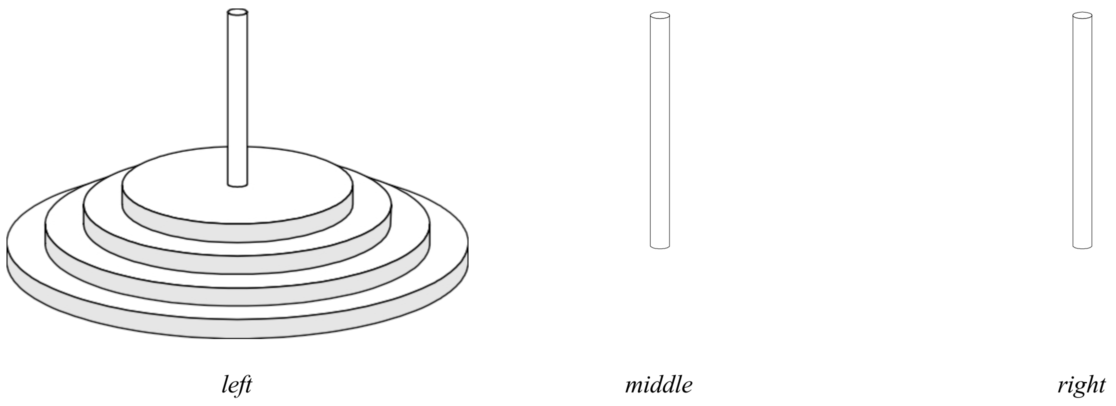


Figure 1: The Towers of Hanoi.

```
fun solve () : nothing

   fun hanoi (rings : int; ref source, target, auxiliary : char[]) : nothing

      fun move (ref source, target : char[]) : nothing
      {
         writeString("Move from ");
         writeString(source);
         writeString(" to ");
         writeString(target);
         writeString(".\n");
      }

   { $ hanoi
      if rings >= 1 then {
         hanoi(rings1, source, auxiliary, target);
         move(source, target);
         hanoi(rings1, auxiliary, target, source);
      }
   } $ hanoi

   var NumberOfRings : int;

{ $ solve
   writeString("Please, give the number of rings:
   "); NumberOfRings < readInteger();
   writeString("\nHere is the solution:\n\n");
   hanoi(NumberOfRings, "left", "right", "middle");
   hanoi(NumberOfRings, "left", "right", "middle");
} $ solve
```

3.3 RAW numbers

The following example program in Grace is a program that calculates the prime numbers between 1 and *n*, where *n is* user-defined. This program uses a simple algorithm to calculate the prime numbers. A pseudo-language formulation of this algorithm is given below. It is assumed that the numbers 2 and 3 are prime, and then only numbers of the form 6k ± 1, where *k* is a natural number, are considered.

MAIN programme

```
print the numbers 2 and 3
for $t$ := 6 to $n$ in steps of 6 do the following:
   if the number $t$ - 1 is prime then print it if the
   number $t$ + 1 is prime then print it
```

Control algorithm (IS number *t* prime?)

```
if $t < 0$ then check the number $-t$ if $t$
$< 2$ then $t$ is not prime if $t = 2$
then $t$ is prime
if $t$ divided by 2 then $t$ is not prime for $i$ := 3 up to
$t/2$ with step 2 do the following:
   if $t$ divided by $i$ then $t$ is not prime o $t$ is prime
```

The corresponding program in Grace is the following.

```
fun main () : nothing

   fun prime (n : int) :
      int var i : int;
   {
      if n <              0then return prime(n);
      else if n <         2then return 0;
      else if n = 2 then return 1; else
      if  n  mod  2  =  0  then  return  0;
      else {
         i < 3;
         while  i  <=  n  div  2  do
            {  if  n  mod  i  =  0
            then
               return 0;
            i < i + 2;
         }
         return 1;
      }
   }

   var limit, number, counter : int;

{ $ main
   writeString("Limit: ");
   limit < readInteger();
   writeString("Primes:\n");
   counter < 0;
   if limit >= 2 then {
      counter < counter + 1;
      writeString("2\n");;
   }
   if limit >= 3 then {
      counter < counter + 1;
      writeString("3\n");;
   }
   number < 6;
   while number <= limit do {
      if prime(number 1) = 1 then {
         counter < counter + 1;
         writeInteger(number 1);
         writeString("\n");
      }
      if number # limit and prime(number + 1) = 1 then {
         counter < counter + 1;
         writeInteger(number + 1);
         writeString("\n");
         writeString("\n");
      }
      number < number + 6;
   }

   writeString("\nTotal:
   ");
   writeInteger(counter);
   writeString("\n");
} $ main
```

3.4 String inversion

The following program in Grace prints the message "Hello world!" by reversing the given string.

```
fun main () : nothing

   var r : char[20];

   fun reverse (ref s : char[]) :
      nothing var i, l : int;
   {
      l < strlen(s);
      i < 0;
      while i < l do {
         r[i] < s[li1]; i
         < i+1;
      }
      r[i] < '\0';
   }

{ $ main
   reverse("\n!dlrow
   olleH"); writeString(r);
   writeString(r);
} $ main
```

3.5 Classification by the bubble method

The bubble sort algorithm is one of the most well-known and simple sorting algorithms. The following program in Grace uses it to sort an array of integers in ascending order. If *x* is the matrix to be sorted and *n* is its size (we assume by Grace convention that its elements are *x*[0], *x*[1], . . *x*[*n-1*]), a variant of the algorithm is described by pseudocode as follows:

Bubble sort algorithm (bubble sort)

```
repeat the following:
    for $i$ from 0 to $n$ - 2
        if $x[i] > x[i + 1]$
            invert $x[i]$ and $x[i + 1]$
as the order of the elements of $x$ varies
```

The corresponding Grace language program is the following:

```
fun main () : nothing

   fun bsort (n : int; ref x : int[]) : nothing

      fun swap (ref x, y : int) : nothing
         var t : int;
      {
         t < x;
         x < y;
         y < t;
      }

      var changed, i : int;
```

```
    { $ bsort
       changed < 1;
       while changed > 0 do
          { changed < 0;
          i < 0;
          while i < n1 do {
             if x[i] > x[i+1] then {
                swap(x[i], x[i+1]);
                changed < 1;
             }
             i < i+1;
          }
       }
    } $ bsort

    fun writeArray (ref msg : char[]; n : int; ref x : int[]) : nothing
       var i : int;
    {
       writeString(msg);
       i < 0;
       while i < n do {
          if i > 0 then writeString(", ");
          writeInteger(x[i]);
          i < i+1;
       }
       writeString("\n");;
    }

    var seed, i : int;
    var x       : int[16];

 { $ main
    seed < 65;
    i < 0;
    while i < 16 do {
       seed < (seed * 137 + 221 + i) mod 101;
       x[i] < seed;
       i < i+1;
    }
    writeArray("Initial array: ", 16, x);
    bsort(16, x);
    writeArray("Sorted array: ", 16, x);;
 } $ main
```

4 Instructions for delivery

The final compiler should be able to export intermediate and final code at will. Unless the `-f` or `-i` function parameters, explained below, are specified, the compiler will accept the source program from a file with any extension (e.g. `*.grc`) given as its unique argument. The intermediate code will be placed in a file ending in `*.imm` and the final code in a file ending in `*.asm`. These files will be in the same directory and have the same main name. For example, the source file `/tmp/hello.grc` will produce the
`/tmp/hello.imm` and `/tmp/hello.asm`.

The executable of the final `gracec` compiler should accept the following parameters:

`-The` optimisation flag (optional).

`-f` program to standard input, output final code to standard output.

`-i` program to standard input, output intermediate code to standard output.

Also, the value returned to the operating system by the compiler should be zero in case of a successful compilation and non-zero otherwise.

For easy development of the compiler it is recommended to use a `Makefile` with the following (at least) features:

- With simple `make`, it will create the executable of the final compiler.
- With `make clean`, it will delete all automatically generated files without exception (e.g. those generated by `bison` and `flex`, object files, but not the final executable).
- With `make distclean`, it will do what `make clean` does, but will also delete the final executable.

An example of such a `Makefile`, assuming that the implementation language is C and the `flex` and `bison` tools are used to produce the Grace language compiler, is the following.

```
CC=gcc
CFLAGS=Wall

gracec: lexer.o parser.o symbol.o general.o error.o symbol.o
        $(CC) $(CFLAGS) o $@ $^ lfl

lexer.c: lexer.l parser.h
        flex s o $@ $<

parser.c parser.h: parser.y
        bison dv o $@ $<

clean:
        $(RM) *.o parser.c parser.h lexer.c core *~

distclean: clean
        $(RM) gracec
```

The display format of the blocks and the final code should be as suggested in the examples. The following guidelines should also be taken into account:

- Intermediate code: adopt a formatting equivalent to

  ```
  printf("%d: %s, %s, %s, %s\n", ...)
  ```

- Final code: attention should be paid to indentation. The following template should be followed:

  ```
  tag: <tab> command <tab> argument1, argument2
       <tab> command <tab> argument1, argument2
  ```