StatPascal
User Manual and Reference

R.-D. Reiss
M. Thomas

Xtremes Group
Zur Alten Burg 29
57076 Siegen
Germany

Inquiries

Please e-mail all inquiries and bug reports to
info@xtremes.de

Check the Xtremes homepage for the latest information:
http://www.xtremes.de

Copyright Notice

Disclaimer

Contents

Chapter 1

Introduction to StatPascal

StatPascal is a statistical programming language that is based on the Pascal language. It is available within the menu system of Xtremes and as a text-oriented commmand line language that runs on Win32 and Unix platforms. StatPascal provides the following extension to standard Pascal that are important within a statistical context.

- Vector and matrix operations,
- a library with statistical routines,
- predefined plots and some simple GUI elements.

In contrast to other statistical languages, StatPascal is strongly typed and employes a compiler to produce code for an abstract stack machine. The resulting runtime performance is usually better than the one achieved with an interpreted system.

StatPascal supports most of the features of the Pascal language to ease porting existing algorithms to StatPascal. However, StatPascal lacks the following facilities of Pascal:

- variant records
- **with** statements
- Pascal–like file access

The first two gaps may be filled in future releases, while the file access is based on the concepts provided by the C language.

This chapter describes the first steps with StatPascal. We start with a description of the integrated StatPascal editor within Xtremes and demonstrate some introductory example programs.

The following typographie is used.

Boldface reserved words within text

Italics identifiers within text

`Typewriter` program listings

Function names specific to StatPascal are written utilizing both lower and upper case letters (e.g. *GaussianDensity*), while functions also available in common programming languages are presented using only lower case letters. StatPascal is not case sensitive, so you can use upper and lower case letters within a StatPascal program.

1.1 StatPascal integrated in Xtremes

A StatPascal editor window is opened by selecting the  button within the toolbar. Please note that under Windows 98, the maximum file size of the editor window is limited to 64 KBytes.

The toolbar enables the user to save and load text files. The *Run* option  is a short cut for the compilation and execution of a program. Table 1.1 lists the available tools.

1.1.1 The 'Hello, World" Program

We start with the traditional 'Hello, world" program to demonstrate the techniques of entering, compiling and running a StatPascal program. The user will immediately recognize that the program looks like its Pascal equivalent.

```
program hello;
begin
    writeln ('Hello, world!')
end.
```

Type in the program, save it to a file (*Save as*,  and click the *Run* button  to execute the program. If the program contains no errors, Xtremes opens the StatPascal window displaying the output.

1.1.2 The StatPascal Window

The procedures *write*, *writeln* and *read* of the Pascal language are provided to perform input and output operations in the StatPascal window. In Fig. 1.1, we see a program in the StatPascal editor window and its output in the StatPascal window.

	New: erases the text within the StatPascal editor window.
	Load: opens the MS Windows file dialog box and loads a StatPascal program.
	Save: writes the text in the StatPascal editor window to a disk file. If no filename has been provided, the *Save as* option is activated.
	Save as: writes the text in the StatPascal editor window to a disk file after asking for a file name.
	Run: compiles and executes the program in the StatPascal editor window.
	Compile: compiles a program and stores the resulting binary under a filename, executes it or locates the position of a runtime error with the source file.
	Compiler Options: opens the *Compiler Options* dialog box, controlling parameters of the compiler and runtime environment.
	Help: opens the StatPascal online help.

Table 1.1: Toolbar of the StatPascal editor

The example program asks for real numbers t and displays the Gaussian density $\varphi(t)$. The predefined routines *MessageBox*, *DialogBox* and *MenuBox* (which are described in the StatPascal manual) provide an alternative using dialog boxes.

One may execute multiple StatPascal programs simultaneously. A new StatPascal window is opened for each running program. By closing the pertaining StatPascal window, a program is aborted.

1.1.3 A second Example Program

The next example adds real numbers entered by the user and prints their sum as soon as a zero is entered:

```
program CalcSum;
(* Program calculating sums *)
var sum, x: real;
```

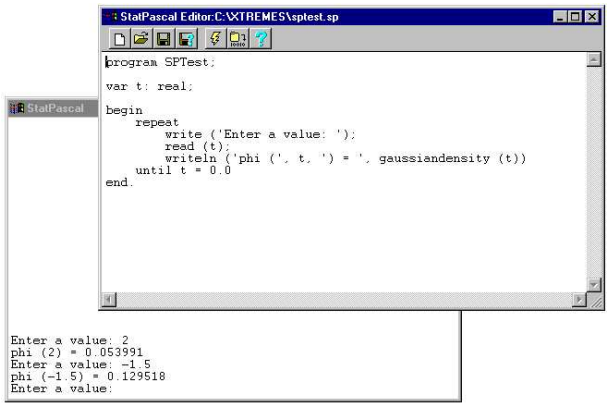


Figure 1.1: StatPascal editor window with example program (front) and its output in the StatPascal window (back).

```
begin
    sum := 0;
    repeat
        write ('Enter number: ');
        read (x);
        sum := sum + x
    until x = 0;
    writeln ('The sum is ', sum)
end.
```

The program contains several statements that are seperated by semicolons. The second line provides a comment. Comments are used to document a program, they do not perform any operations.

The third line shows a variable declaration: `var sum, x: real`. We provide two variables (*sum* and *x*) that are used to store real numbers. Within the program, *sum* contains the sum of the numbers entered previously. In the first statement (given on the fourth line), sum is initialized with 0. Then the program enters a loop (**repeat-until**) that is executed until the condition at the end of the loop (`x = 0`) is satisfied. Within the loop, the user is prompted for numbers that are added to sum. When the user inputs 0, the loop terminates, and the sum of the values is displayed.

Chapter 2

Programming with StatPascal

The beginnning of this chapter addresses the novice programmer who has little or no experience with programming languages. We provide an introduction to the StatPascal language that can hopefully be read without prior knowledge.

This chapter assumes that you are familiar with the tasks of entering and executing a program as shown in the previous chapter. Experienced Pascal programmers may start with the section Differences between StatPascal and Pascal (see section A.1). An important extension for statistical programming is the vector structure (see Section 3).

2.1 Basic Programming

2.1.1 Variables and Data Types

Variables are used to store values that change during the execution of a program, e.g. values entered by the user or results of calculations. A variable must be declared before it can be used. Each variable has a data type that determines a set from which values may be stored. We mention some basic data types.

integer	integer numbers	7, 5
real	real numbers	2.71828, -3.12e-15
char	single characters	'a', '7'
string	sequence of characters	'A string'
boolean	logical values	false, true

The datatype *string* does not belong to standard Pascal, but is supported by many implementations of the language. The distinction between *integer* and *real* values is made because of the internal representation of these values. Operations

with real values are subject to rounding errors, which is not the case with integer operands. In the following, we use *type* to represent any data type.

Variables are declared at the beginning of a program by means of a variable declaration taking the form

var *identifier*, *identifier*, ... : *type*;

The word *identifier* is used for names of variables (and also procedures, functions, etc. introduced later) that are provided by the programmer. The following example shows the declaration of two *integer* and a *real* variable:

```
var i, j: integer;
    x: real;
```

2.1.2 Expressions and Assignments

After declaring a variable, a value is assigned to it using the statement

identifier := *expression*

Here, the variable *identifier* is set to the value of the expression. Expressions are built using numbers, variables, functions and operators combining them. The following example shows some assignments.

```
var a, b: integer;
    x, y: real;
begin
    a := 2;
    b := 5 * a + 3;
    x := 3.1415;
    y := sin (x) - 1.5;
    y := 2 / (x + 1)
end.
```

In the above expressions we used addition (`+`), substraction (`-`), multiplication (`*`), division (`/`) and the function *sin*. The operators `*` and `/` have a higher priority than `+` and `-`, so brackets were used in the last example.

StatPascal is a typed language, and the compiler checks if the types used in an expression or in an assignment are the same. Some implicit type conversions are provided, e.g. it is possible to assign an integer value to a real number.

In our above example, `a := x` would result in an error since a real number cannot be assigned to an integer. It is possible to perform an explicit conversion by means of the functions *round* (rounding a value to the nearest integer) and *trunc* (truncating the fractional part, thus `trunc (3.14) = 3` and `trunc (-2.71) = -2`).

StatPascal also works with logical values: A data type *boolean* with the values *true* and *false* is provided. These values occur as the result of comparisons (recall the condition `x = 0` in the second example). Other relational operators are `<`, `>`, `<=`, `>=` and `<>` (not equal). Moreover, the logical operators **and**, **or**, **not** and **xor** (exlusive or, not available in standard Pascal). are available. The following example shows the use of logical expressions.

```
var a, b, c: boolean;
    x, y: integer;
begin
    x := 2;
    y := 3;
    a := true;
    b := false;
    c := a and b;             (* false *)
    c := a or b;              (* true *)
    c := x = 2;               (* true *)
    c := x > y;               (* false *)
    c := not a;               (* false *)
    c := (x < y) and (x = 2)  (* true *)
end.
```

The **and**-operator has a higher priority than the relational operators, so brackets are required in the last assignment. Although logical expressions are most commonly used in conditional statements (like **repeat - until**), it is possible to assign the results of these expressions to variables of the data type *boolean*.

2.1.3 Constant Declarations

Constants are declared by means of a constant declaration

const *identifier* = *value* **;**

The example shows a constant declaration

```
const n = 50;
      Pi = 3.1415;
      s = 'String constant';
```

In the Pascal language, such a declaration must appear before the variable declaration. StatPascal allows an arbitrary mix of declarations. Note that the `=` operator is used to assign a value to a constant, while `:=` is used in conjunction with variables. The following example defines the constant `Pi` and uses it in an expression.

```
program constdemo;
const Pi = 3.1415;
var x: real;
begin
    x := sin (Pi);
    writeln ('sin (Pi) = ', x)
end.
```

2.1.4 Input and Output

The input and output operations of StatPascal resemble a terminal-like interface. One may write values to an output device (a text window if StatPascal is embedded in a graphical environment or standard output) using *write* and *writeln*. The values must be given in brackets and are seperated by commas. The following example shows some *write* and *writeln* statements.

```
var a, b: real;
begin
    write (2.71);
    write (a, 2 * b + 0.5);
    write ('sin(a)=', sin (a));
    writeln;
    writeln (a = b)
end.
```

writeln performs the same operations as *write*, then it advances to the next line. Logical values (like `a = b`) are printed as `true` or `false`.

You can read values by means of a *read*-statement. When a *read*-statement is performed, the computer waits for you to type in the values requested. If more than one value is needed, they are separated by blanks when typed in. Character strings should be the last item in a *read*-statement since any text up to the end of the line (which is entered by pressing the return key) is stored in the string. The following program performs a simple dialog by means of *read* and *write* statements.

```
program sum;
var s: string;
    a, b: real;
begin
    write ('Enter your name: ');
    read (s);
    write ('Enter two numbers: ');
    read (a, b);
    writeln ('Hello, ', s);
    write ('The sum of ', a, ' and ', b, ' is ', a + b)
end.
```

2.1.5 Conditional statements

Conditional statements control the execution of statements depending on a condition.

2.1.5.1 if-statement

The **if-then-else** statement has the general form

if *expression* **then** *statement* [**else** *statament*]

If the (logical) expression is true, then the statement following **then** is executed. One may also provide an optional **else**-part that is executed if the condition is *false*. Some examples are:

```
if x = 0 then
    writeln ('x = 0');

if (a < b) and (c = 5) then begin
    d := 7; e := 2
end else
    x := x + 1;

if a = b then
    if c = d then
        e := 5
    else
        f := g;
```

One should note the following details:

- A semicolon in front of **else** is not allowed.
- Only one statement may be given in the **then** or **else** part. However, two or more statements can be grouped together using **begin** and **end**. Such a *compound statement* may be used whenever a single statement is required. Within the compound statement, the single statements are seperated by semicolons. It is not necessary to provide a semicolon in front of **end**.
- **if**-statements may be nested (see the third example). An **else** belongs to the most recently used **if**- statement. One may use **begin** and **end** to enforce another grouping; thus, if you write

```
if a = b then begin
    if c = d then
        e := 5
end else
    f := g;
```

then `f := g` is executed if the condition `a = b` is false.

2.1.5.2 case-statement

The **case**-statement is used to select one option from several alternatives. The following example determines the length of a month:

```
case month of
    1, 3, 5, 7, 8, 10, 12: length := 31;
    4, 6, 9, 11:           length := 30;
    2:                     length := 2
    else                   length := -1   (* error *)
end;
```

The result of the expression is compared with the constant values provided, the statement with a matching label is executed. Note that in standard Pascal a program terminates with a run time error if no match is found, while StatPascal ignores the **case**-statement or executes the optional **else** branch.

2.1.6 Loops

StatPascal provides three different loop constructs which allow the repeated execution of a statement.

2.1.6.1 while-loop

The **while**-loop repeats a statement as long as a given condition is satisfied. The condition is checked before each iteration of the loop, and the entire loop is skipped if the condition already fails when the program enters the loop. The general form is given by

while *condition* **do** *statement*

Only one statement may be controlled by the loop. Two or more statements must be grouped together using **begin** and **end**, thus forming a compound statetment. The following example adds the values from 1 to 100 using a **while**-loop.

```
program sum2;
var i, sum: integer;
begin
    sum := 0;
    i := 1;
    while i <= 100 do begin
        sum := sum + i;
        i := i + 1
    end;
    writeln (i)
end.
```

2.1.6.2 repeat-loop

The **repeat**-loop repeats a sequence of statements given between **repeat** and **until**, it terminates when the condition given after **until** is satisfied. The condition is checked after the statements are executed, so they are performed at least once even if the condition is already fulfilled at the beginning of the loop. The general form of the **repeat**-loop is

repeat *statement* {*;statement*} **until** *condition*

The following example calculates the sum of the numbers 1 to 100 using a **repeat**-loop:

```
program sum1;
var i, sum: integer;
begin
    sum := 0;
    i := 1;
    repeat
        sum := sum + i;
        i := i + 1
    until i > 100;
    writeln (i)
end.
```

Note the following details:

- No **begin-end** is required to group the statements within the loop.
- A semicolon in front of **until** is not necessary.

2.1.6.3 for-loop

The **for**-loop is used to repeat a statement a predefined number of times. It takes the following general form:

for *identifier* **:=** *expression* **to** *expression* **do** *statement*
for *identifier* **:=** *expression* **downto** *expression* **do** *statement*

The statement is repeated with increasing (or decreasing if **downto** is employed) values of the *control variable* specified by *identifier*, starting with the value given by the first expression. After each execution of the loop, the variable is incremented (decremented) by one, and the loop is repeated if the variable is less (greater) than or equal to the value provided by the second expression. The control variable must be of an integer type (or an ordinal type introduced later), and the two expression must yield results of that type. The example calculates the sum of the numbers from 1 to 100 using a **for**-loop.

```
program sum3;
var i, sum: integer;
begin
    sum := 0;
    for i := 1 to 100 do
        sum := sum + i;
    writeln (sum)
end.
```

We mention some details:

- If the second expression yields a result that is smaller (larger) than the start value given by the first expression, the entire loop is skipped.
- Two or more statements inside a **for**-loop must be grouped using **begin** and **end**.
- The expressions providing start and end value are evaluated once upon entering the loop. If their values change during the execution of the loop, then the number of repetitions of the loop is not affected. In the following example, the loop is repeated five times:

```
var i, e: integer;
begin
    e := 5;
    for i := 1 to e do begin
        e := 1;
        ...
    end
end.
```

- Standard Pascal does not allow changes to the control variable within the loop by means of assignments, which is legal in StatPascal.

2.1.7 Procedures and Functions

Procedures and functions are used to group together operations and make them accessible from other parts of a program. If a operation is to be performed more than once, it can be defined as a procedure or function that is called from the places where it is needed.

Functions (like the predefined function *sin*) return a value that is used within an expression, while a procedure (like *writeln*) is used as a statement. Both kinds of subroutines are defined after the variable declarations, and both may take parameters provided by the caller.

The following example shows the declaration of a function and a procedure:

```
program demo;

function square (x: real): real;
    begin
        square := x * x
    end;

procedure table;
    var i: integer;
    begin
        for i := -10 to 10 do
            writeln (i/10, '  ', square (i/10))
    end;

begin
    table
end.
```

The header of a subroutine consists of the reserved word **function** or **procedure** followed by its name. An optional parameter list is included in brackets, and consists of one of more identifiers, seperated by commas and followed by a colon and the data type associated with the parameters. Two or more such parameter blocks may be specified, they are seperated by semicolons. A function requires a return type that is given as the last part of the header: a colon followed by the name of the returned type. Thus, the following headers are valid:

```
procedure proc1 (a, b: integer; c, d: real);
function func1 (a, b, c: string): integer;
function func2: real;
```

The last line contains a function taking no parameters at all. Such a function is still useful since it can address global variables within the program.

The body of a subroutine is structured like a program: one may declare constants, variables and further procedures and functions. The statements performed by the subroutine are given between **begin** and **end**. A subroutine ends with a semicolon.

Identifiers declared within a subroutine are local, they cannot be accessed from the outside. Thus, the variable *i* declared within *table* in the above example can only be used within *table*. If you declare an identifier that already exists outside the function, then the original identifier is hidden inside the routine; its original meaning is restored at the end of the routine.

Functions and procedures may call themselves recursively. One may implement a function calculationg the faculty in the following way.

```
function fac (n: integer): integer;
```

```
    begin
        if n = 0 then fac := 1
        else fac := n * fac (n - 1)
    end;
```

The parameters of a subroutine are treated like local variables. Assigning values to them does not affect the variables that were given in the call of the subroutine. If you want to change the original values, then the parameter declaration must be proceeded by **var**. Consider the following example:

```
program test;
var n: integer;

procedure assign (var j: integer);
    begin
        j := 6
    end;

begin
    n := 1;
    assign (n);
    writeln (n)
end.
```

This program prints 6 because n is passed as a variable parameter to *assign*. It is not possible to provide an expression (like `3.2*sin(x)`) when a variable parameter is required.

2.1.8 Forward Declarations

A problem arises when two subroutines must call each other. Because StatPascal requires that identifiers are declared before they can be used, the subroutine that is implemented first does not have access to the second one.

As a solution to this problem, it is possible to only declare the name and parameter types of a subroutine. Instead of giving an implementation of the routine after the function or procedure header, one writes the reserved word **forward**, as in the following (technical) example.

```
procedure p1 (n: integer); forward;

procedure p2 (n: integer);
    begin
        if n < 10 then writeln (n)
        else p1 (n)
    end;
```

```
procedure p1;
    begin
        if n >= 10 then writeln (n)
        else p2 (n)
    end;
```

One can see that `p1` calls `p2` and vice versa. By declaring `p1` as forward, `p2` has enough information about `p1` to perform the call to it. Note that the parameter list of `p1` is not repeated when the procedure is actually defined.

The same holds for the definition of function that was declared forward: in that case one also must omit the return type and just give the function name, as, e.g.

```
function f (x: real): real; forward;
...
function f;
    begin
        f := 2 * x
    end;
```

Functions and procedures defined in the interface part of a unit (see Section 4.1) are declared forward implicitly.

2.2 Data Structures and Types

StatPascal provides the user with data structures that allow the organization of data in various ways. The most important structure is the array. Readers who are unfamiliar with the concepts presented in this section may skip the other structures on the first reading.

2.2.1 Arrays

Arrays enable the programmer to store multiple values of the same data type within one variable and to access them utilizing an index that is calculated at run time. To declare an array, one must provide a lower and upper bound for the index as well as the type of the values to be stored. The following example shows some array declarations.

```
var x: array [1..100] of real;
    b: array [-7..15] of boolean;
    a: array [1..10] of array [1..15] of real;
```

An array is a new data type, so one may declare arrays that consist of other arrays. The variable `a` in the above example can be considered as a real matrix with 10 rows and 15 columns. Such declarations can be given in the shorter form

```
var a: array [1..10, 1..15] of real;
```

The components of an array are accessed by means of an index given in square brackets. With the above declarations, the following is legal.

```
for i := 1 to 100 do
    x[i] := cos (i / 100 * Pi);
b[-2] := false;
a[i][j] := 0.8;
a[3,2] := 1.2;
a[3] := a[2];
```

Note that `a[3,2]` is a shorthand notation for `a[3][2]`. The last assignment shows that entire arrays of the same type (in this case `array [1..15] of real`) can be copied.

2.2.2 Type declarations

StatPascal allows the user to define names for data types by means of a type declaration. Such a type declaration is required if you want to declare (and assign) variables of a user defined type at different locations within a program. For example, one might introduce two arrays by means of the declaration

```
var a: array [1..10] of real;
    b: array [1..10] of real;
```

Although `a` and `b` have the same structure, they are considered to belong to different types since the array type is constructed twice. Therefore, the assignment `a := b` is illegal. Using the declarations

```
type field = array [1..10} of real;
var a: field;
    b: field;
```

the assignment `a := b` becomes legal.

A type declaration is necessary if you want to provide an array as argument to a function or procedure. We provide an example of a routine calculating the mean of a real array. Note that a type `field` is used to declare an array in the procedure `demo` and to pass it to the function `mean`.

```
const n = 100;

type field = array [1..n] of real;

function mean (a: field): real;
    var sum: real;
```

```
        i: integer;
    begin
        sum := 0.0;
        for i := 1 to n do
            sum := sum + a [i];
        mean := sum / n
    end;

procedure demo;
    var a: field;
        i: integer;
    begin
        for i := 1 to n do
            a [i] := i;
        writeln (mean (a))
    end;

begin
    demo
end.
```

A major drawback of this approach is the requirement to define the size of the array (100 in the present case) as a constant within the program. StatPascal provides vector types (see Section 3) as extensions to the Pascal type system whose sizes are defined at runtime.

2.2.3 Ordinal types

Ordinal types have values within a countable set. StatPascal provides the predefined ordinal types *boolean*, *char* and *integer*. Note that *real* is not an ordinal type. A new ordinal type is introduced by means of enumerations and subrange declarations.

2.2.3.1 Enumerations

A simple way to introduce a data type is to enumerate all its values, e.g.

```
type color = (red, green, blue, black);
```

The identifiers used in the list are treated as constants of the type declared. An order is defined on the type based on the position of the values in the list (e.g. `red < blue`). The values are converted to integers by means of the function *ord*. *ord* returns the position of the value in the list, starting with 0 (i.e. `ord (red) = 0`, `ord (green) = 1`, etc.).

One may use enumerated types as an index of arrays. Given the above declaration, the following is valid.

```
var a: array [color, 3..5] of integer;
    b: array [green..black] of real;
```

Moreover, variables of enumeration types may be used as control variables of **for**-loops.

```
var c: color;
    d: array [color] of integer;
begin
    for c := red to black do
        d [c] := ord (c)
end.
```

2.2.3.2 Subranges

One may declare a new data type as a subset of another ordinal data type. We provide some examples.

```
type month = 1..12;
     day = 1..31;
     color = (red, green, blue, black);
     mycolor = green..blue;
```

Whenever a value is assigned to a subrange type, StatPascal checks if the value is within the specified range and terminates the program if the check fails. For example, the assignment `dayvar := 25` is valid, but `dayvar := 32` fails if `dayvar` is a variable of type `day`. Subranges do not only document a program, but also enforce a check on the values assigned to a variable.

Note that these checks can be turned of in the Compiler Options dialog (see 4.7.3) to increase the execution speed of a program.

2.2.4 Records

Records are used to group together data belonging to the same or different data types. Instead of indexing them with a number, the components are addressed by means of individual names. A record type storing complex numbers is defined by

```
type complex = record
                   re, im: real
               end;
```

Having declared a variable of the above data type, its components are accessed by appending the component name (i.e. `re` or `im` in our example) to the variable name:

```
var c, d, e: complex;
begin
    c.re := 0;
    c.im := 1;
    d.re := -c.re + 2;
    e := c
end.
```

Arrays of record types may be built, and records can in turn contain components that are arrays. The following example shows a declaration that allows the storage of data collected from a student during a course: His name, delivered homeworks and points achieved in examinations.

```
type student = record
                   name: string;
                   homework: array [1..12] of integer;
                   examination: array [1..2] of integer
               end;

var class: array [1..45] of student;
```

Now, to store the points achieved by the fifth student in his third delivered homework, one would write

```
class[5].homework[3] := ...
```

2.2.5 Sets

StatPascal handles sets of ordinal data types. The following example shows the declaration of two variables, storing a set of the values 1 to 20:

```
var a, b: set of 1..20;
```

A set is defined by listing its members in square brackets:

```
a := [2, 7, 13];
b := [1, 2, 3, 5];
```

The empty set is denoted by `[]`. One may unite sets by means of the operator `+`, `*` calculates the intersection and `-` the difference set. The boolean operator **in** is used to check if a value is a member of a set, e.g. `2 in [2, 7, 13]` is true. The number of values of the base type of a set is limited to 256; thus, the declaration `set of 1..500` is illegal. The reader should note that this limitation may vary between different implementations of Pascal.

2.2.6 Pointers

Pointers are used to store references to other objects that are created at run time. A procedure *new* is provided to create an object and to initialize a pointer to it. We give an example of this data type:

```
var a: ^integer;
...
new (a);
a^ := 42;
write (a^);
dispose (a);
```

The pointer type is introduced by means of `^`. In the above example, `a` is a variable that contains a pointer to an integer object. The call to new creates an integer object and assigns a pointer to it to `a`. The assignment `a^ := 42` stores a value within that object. It should be released using a call to `dispose` when it is no longer needed.

Pointers are mainly used to create dynamical data structures (like lists and trees). A simple example of such a structure is a linear list. Each list entry contains the data one wants to store, and a pointer to the next entry. Since the list is build at run time, there is (theoretically) no limit on the amount of data one is able to store. The type declaration for a list looks like

```
type nodeptr = ^node;
     node    = record
                   data: integer;
                   next: nodeptr
               end;
```

Note that it is possible to declare a pointer to a data type declared later on, so the above recursive definition is legal.

Pointers provide a powerful mechanism to implement intricate data structures like trees or graphs. The reader is referred to an advanced book (e.g. Wirth, N.: *Algorithms and Data Structures*) for details.

Chapter 3

Vector and Matrix Operations

StatPascal implements the new data structures **vector** and **maxtrix** which are similiar to the array structure; yet, one does not have to specify the number of elements when declaring a vector or matrix. Such a type is defined using the declaration

vector of *type*

or

matrix of *type*.

Vectors are initialized with an empty vector upon their creation. StatPascal deallocates the memory used by a vector automatically and provides a compactification of the heap where these objects are stored within its run time environment.

In the following sections, the operations that are supported for vectors and matrices are described, and examples are given. We start with the vector structure.

3.1 Introductory Examples

We start with a simple example that shows the usage of a real vector. The following program generates a Gaussian data set with location parameter 2 and scale parameter 3 and stores it in the vector `x`. It then displays the mean and variance of the simulated data set.

```
program example;
var x: vector of real;
begin
```

```
    x := 2 + 3 * GaussianData (100);
    writeln (mean (x), variance (x))
end.
```

Readers who are familar with other statistical languages should note that the usual arithmetic and logical expressions (with componentwise operations) as well as index operations are supported. Vectors can also be used as arguments and return types of functions. The language provides implicit looping over the components of a vector if a function operates on the base type of a vector structure. Details on these topics are given in the following sections.

The next program demonstrates further vector operations. We perform a numerical integration of a function $f : \mathbb{R} \to \mathbb{R}$ using the approximation

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{n} \frac{f(c_i) + f(c_{i+1})}{2} \frac{b-a}{n},$$

where $c_i = a + (i-1)(b-a)/n$, $i = 1, \ldots, n+1$. The function f that is integrated as well as the parameters a, b and n are provided as arguments.

```
type realfunc = function (real): real;

function integrate (f: realfunc; a, b: real; n: integer): real;
    var fc: vector of real;
    begin
        fc := f (realvect (a, b, n + 1));
        return sum (fc [1..n] + fc [2..(n+1)]) * (b-a) / (2*n)
    end;
```

We start with a type declaration for the functional parameter (see section 4.2 for details on functional parameters). The first assignment within the function *integrate* calculates the values $f(c_i)$, $i = 1, \ldots, n+1$ and stores them in the variable *fc*. Note that the call to the predefined function *realvect* returns a real vector with $n+1$ equally spaced points between a and b, which is given as an argument in the call of *f*.

In the second statement, we generate two integer vectors containing the values from 1 to n and from 2 to $n+1$, which serve as indices to *fc*. The index operation yields two real vectors with the values $(f(c_1), \ldots, f(c_n))$ and $(f(c_2), \ldots, f(c_{n+1}))$. The $+$ operator adds these vectors componentwise, and the predefined function *sum* calculates the sum of the components of the resulting vector. Finally, the value of the integral is returned.

Next, we define a function `square` and calculate its integral.

```
function square (x: real): real;
    begin
        return x * x
```

```
    end;

begin
    writeln (integrate (square, 0, 1, 100))  (* 0.33335 *)
end.
```

3.2 Construction of Vectors

A value of a vector type is obtained in one of the following manners.

1. The **..** operator constructs an *integer* vector, i.e. m**..**n returns a vector with the values $m, m+1, \ldots, n$. If $m > n$, then an empty vector is returned. S-Plus users should note that we cannot use the **:** operator because it is utilized in *write* to format the output.

2. The predefined function *realvect (a, b, m)* returns a vector d of *real* values with $d_i = a + (i-1)/(m-1)(b-a), i = 1, \ldots, m$.

3. The *combine* function constructs a vector from an arbitrary number of arguments, which must be vectors or single values of the same type t. The usual type conversions (e.g. from *integer* to *real*) are supported. The type of the vector is defined by the type of the first argument (after discarding subranges and performing calls to functions without parameters).

Example: (Construction of Vectors.) The following example illustrates the construction of a vector.

```
var a: vector of integer;
    b: vector of real;
begin
    a := 3..15;
    b := realvect (0, 1, 11);
    a := combine (2..4, 8, 13..15)
end.
```

The first statement generates an integer vector with values from 3 to 15, while the second one produces a real vector with the values $0.0, 0.1, \ldots, 1.0$. The last statement utilizes the *combine* function to generate an integer vector with the values $2, 3, 4, 8, 13, 14, 15$. Note that the assignment frees the memory allocated for the vector `3..15`.

3.3 Type Conversions

StatPascal supports the following type conversions related to vectors.

1. An array of type t can be converted to a vector of type t.

2. A vector of *integer* values can be converted to a vector of *real* values.

3. A value of type t can be converted to a vector of type t.

These conversions allow the application of all vector operations to arrays and the mixing of scalar values and vectors in arithmetic expressions.

3.4 Assignments

Vectors having the same base type can be assigned to each other, regardless of the number of elements. In particular, vectors originating from different type declarations are compatible as long as their base type is the same, as e.g.,

```
var a: vector of real;
    b: vector of real;
...
    a := b;
```

The relaxed type compatibility is required because we do not provide constructors to specify a type when a vector is generated. Standard Pascal uses a similiar mechanism for set construction.

3.5 Output

Vectors of the types *integer*, *char*, *real* and *boolean* are allowed as arguments of a call to the *write* procedure.
Example: (Printing Vectors.) The following example prints the elements of the vectors given in the call of the *writeln* procedure, seperated by a blank.

```
begin
    writeln (3..15, 7..20)
end.
```

The format options of the Pascal *write* procedure (one or two additional values seperated by colons) are also available for vectors, and a second value (or a third one for real vectors) defines the number of columns written before a line feed is emitted. Thus, the statement `writeln (a:10:5:6)` would print the real vector a using 6 columns, each 10 characters wide with 5 digits after the decimal point.

3.6 Expressions

The arithmetic operators (+, −, ∗, / and ∗∗) and relational operators (<, <=, >, >=, = and <>) can be applied to vectors of the types *integer* and *real*. Numerical operators yield a real vector if one of the operands is a real vector. The relational operators return a boolean vector.

When applying the operator $\otimes$ (with $\otimes$ being one of the above operators) to the vectors a and b with length m and n, the resulting vector c is defined by

$$c[i] := a[(i-1)\text{mod } m+1] \otimes b[(i-1)\text{mod } n+1], 1 \leq i \leq \max(m,n).$$

Note that the shorter vector is repeated until the length of the longer one is reached when the vectors have different length. This semantic was chosen because it allows a natural combination of vectors and scalars: An expression like `2 * a` (where a is a vector) is handled by converting the scalar value to a vector with the length 1, and then the multiplication is applied to all components of a.

The data type *string* is implemented as **vector of** *char* and gets a different treatment in comparisons; see section 3.10 for details.

Example: (Applying Operators to Vectors.) The boolean vector obtained by the expression `a < b` is printed using 'FALSE' and 'TRUE'.

```
var a, b: vector of integer;
begin
    a := 1..10;
    b := combine (1..5, 9..13);
    writeln (a + b);
    writeln (a < b)
end.
```

3.7 Index Operations

Given a vector a with *size (a)=l* (denoting the number of elements), the following index operations are possible:

$a[n]$	n *integer*; returns the n-th component of the vector a, $1 \leq n \leq l$. The returned value is of the base type of the vector.
$a[b]$	b **vector of** *boolean* with *size(b)* $\leq l$, extracts the components $a[i]$ for which $b[i]$ is *true*. The returned vector has the same type as a.
$a[c]$	c **vector of** *integer* with $1 \leq c[i] \leq l$ for $1 \leq i \leq$ *size(c)*, yields a vector d with the values $d[i] = a[c[i]], 1 \leq i \leq size(c)$. The returned vector has the same type as a.

These index operations do not produce l-values, i.e. it is not possible to use them on the left side of an assignment.

Example: (Index Operations.) We demonstrate the index operations.

```
var a, b: vector of integer;
begin
    a := 1..10;
    b := 4..9;
    writeln (a [3]);              (* 3 *)
    writeln (a [a > 5]);          (* 6 7 8 9 10 *)
    writeln ((2 * a)[b div 2])    (* 4 4 6 6 8 8 *)
end.
```

The *writeln* statements use the three possible index operations. The first one employs an integer value as index. The second one uses a vectorized comparison to obtain a boolean vector with the index positions where a exceeds 5. The last operation uses an integer vector listing the positions to be extracted. Note that the index operations of vectors can be applied to vector expressions (like `2 * a`) and not just variables.

3.8 Function Calls

A vector a of integer or real values can be given as argument to the call of a function $f : \mathbb{R} \to \mathbb{R}$ or $f : \mathbb{N} \to \mathbb{N}$. The resulting vector is defined by $f(a[i]), i = 1, \ldots, size(a)$.
Example: (Function Calls.)

```
function f (x: real): real;
    begin
        f := 2 * x
    end;

begin
    writeln (sqr (1..5));                    (* 1 4 9 16 25 *)
    writeln (f (realvect (-0.5, 1.5, 5)))    (* -1 0 1 2 3 *)
end.
```

The first call passes the integer vector `1..5` to the predefined square function, while the second call passes a real vector as argument of a real-valued function. A loop that calls the function for the single values is generated by StatPascal when a user defined function is called.

3.9 Special Operations

The following functions can be applied to any vector type.

1. *size* returns the number of components of a vector.
2. *rev* reverts the order of components of a vector.

It is not possible to implement these functions using the StatPascal language, i.e. there is no way to obtain the size of an arbitrary vector other than by calling the *size* function. These limitations exist because it is impossible to write a function that accepts vectors of any type.

3.10 Strings

The data type *string* is predefined as a character vector. A constant of the string type is specified in the usual way (see below for some examples). Applying relational operators to a string type does not perform a vectorized comparison of the single characters as described in section 3.6. Instead, an alphabetical comparison is performed. Moreover, the + operator concatenates two strings.

The special treatment is only applied to values of the predefined *string* type, not to other character vectors. There is no type conversion operator, but one can use assignments to the appropriate type to obtain the desired treatment. No maximum length for strings exists.

Example: We demonstrate the differences between strings and character vectors.

```
var a, b: string;
    c, d: vector of char;
begin
    a := 'Hello, world'; b := 'Hello';
    c := 'AGM'; d := 'ABY';
    writeln (a > b);        (* TRUE *)
    writeln (c > d);        (* FALSE TRUE FALSE *)
    writeln (a [1..5]);     (* Hello *)
    writeln (size (a))      (* 12 *)
end.
```

The first comparison yields a single logical value because it is applied to a string, while the second one produces a boolean vector. All vectorized index operations are applicable to strings and return a string as their result; the *size* function determines the length of a string.

3.11 Matrix Operations

The data structure **matrix** represents two–dimensional arrays where the number of rows and columns are determined at run time. The language provides an implicit conversion from two–dimensional arrays to matrices. Thus, it is possible to provide an array whenever a matrix is expected, e.g., in calls to predefined functions and procedures.

One can also construct a matrix using predefined functions. MakeMatrix (see Section 7.97) fills a matrix with the components of a vector, while UnitMatrix (see

Section 7.191) creates a unit matrix. The following examples demonstrates these possibilities.

```
var A: matrix of real;
begin
    A := MakeMatrix (1..6, 2, 3);
    writeln (A); (* 1 2 3 *)
                 (* 4 5 6 *)
    A := UnitMatrix (3);
    writeln (A);    (* 1 0 0 *)
                    (* 0 1 0 *)
                    (* 0 0 1 *)
end.
```

Matrices can be used in arithmetic operations. The multiplication of two matrices or of a matrix and a vector perform the usual mathematical matrix operations. The predefined functions Transpose (see Section 7.185) transposes a real matrix, while Invert (see Section 7.92) calculates the inverse matrix. Another function is Chol (see Section 7.18) which performs the Cholesky decomposition. The following example uses this function to simulate Gaussian random variables with a given covariance matrix.

```
program bivgauss;
var S, C: matrix of real;
    i: integer;
begin
    S := MakeMatrix (combine (1.0, 0.2, 0.2, 1.5), 2, 2);
    C := chol (S);
    for i := 1 to 100 do
        writeln (C * GaussianData (2))
end.
```

Chapter 4

Advanced StatPascal Techniques

4.1 Units

Units are used to modularize programs by breaking them into several smaller parts and to implement libraries of predefined routines that can be called from different programs. We start with an example that shows a simple unit.

```
unit example;

interface

procedure hello (s: string);

implementation

procedure hello;
    begin
        writeln ('Hello, ', s)
    end

end.
```

After defining the unit name (`example`), an interface and an implementation section are specified. The interface section declares all constants, types, variables, procedures and functions that are provided by the unit. In the above example, this is the procedure `hello`. The definition of the procedure takes place in the implementation section. Note that — similar to a forward declaration — the parameter

list is not repeated when the procedure is actually implemented (doing so would result in an error message).

The unit must be stored in a file with the unit name and the extension .sp. In the above example, one would have to store the unit `example` in the file `example.sp`.

To utilize a unit in a program, it is imported after the definition of the program name with the reserved word **uses**. The following program imports the unit `example` and calls the procedure `hello`.

```
program demo;
uses example;
begin
    hello ('Name')
end.
```

The separation between interface and implementation section enables a unit to make private declarations (e.g., variables or further subroutines) that are not visible outside the unit. The interface section may also start with a **uses** declaration to import further required units.

The next example shows a unit performing a numerical integration of a function given as argument. It also utilizes functional parameters (see Section 4.2) introduced in the next section. An example unit with a more sophisticated integration algorithm is provided in the `sp`–subdirectory of the Xtremes installation.

```
unit integration;

interface

type realfunc = function (real): real;
function integrate (f: realfunc; a, b: real; n: integer): real;

implementation

function integrate;
    var fc: vector of real;
    begin
        fc := f (realvect (a, b, n + 1));
        return sum (fc [1..n] + fc [2..(n+1)]) * (b-a) / (2*n)
    end;

end.
```

Multiple units can be included by separating their names with commas. The compiler searches the corresponding files within the working directory and within the directories listed in the *Search Path for Units* in the *Compiler Options* dialog box

4.2 Functional and Procedural Types

StatPascal supports procedural and functional data types, i.e. one may declare variables that store references to functions and procedures, e.g.

```
type realfunc = function (real): real;

var g: realfunc;

function f (x: real): real;
    begin
        f := 2.0 * x
    end;

begin
    g := f;
    writeln (g (3.0))      (* 6 *)
end.
```

Two subroutines are of the same type if they expect the same number and types of arguments and if they return values of the same type in the case of a function. Such types are important because they allow the provision of functions as arguments of subroutines.

Using functional or procedural variables, it is possible to make subroutines visible outside their scope. Performing calls to such subroutines may lead to errors if the subroutine accesses nonlocal variables.

4.3 Evaluation of Boolean Operands

StatPascal offers a shortcut evaluation for the boolean operators **and** and **or**. Contrary to Pascal, all operands are guaranteed to be evaluated from left to right. If the option *Boolean Shortcuts* is selected in the *Compile* dialog box, then the evaluation of an **and** operator is terminated if the first operand evaluates to *false*. Likewise, the second operand of the **or** operator is not evaluated if the first one returns *true*. Expressions like `(x > 0) and (log (x) < c)` are, therefore, possible (this code breaks in Pascal because it is tried to evaluate the logarithm even if $x \leq 0$ and the expression is false anyway).

As an example, consider the search for the smallest value exceeding a threshold t. Using the shortcut evaluation of boolean operators, the search is performed by

```
i := 1;
while (i <= samplesize) and (data (i) <= t) do i := i + 1;
```

Without the shortcut evaluation, one has to write

```
i := 1; Done := false;
while (i <= samplesize) and not Done do
    if data (i) > t then Done := true
    else i := i + 1;
```

to stop the system from accessing `data (samplesize + 1)` if no value exceeds the threshold t.

4.4 Generating and Accessing Data

An important facility of StatPascal is the implementation of routines for data generation and data transformation not covered by the menu system or UserFormula. In this section, we introduce functions and procedures to exchange data between StatPascal and Xtremes. Data stored in a StatPascal program (e.g., in a vector) are not used directly within Xtremes, and data sets loaded in Xtremes are not used by StatPascal automatically. Instead, all data transfer is accomplished by calling predefined functions and procedures. They give the user access to the active data set from within a StatPascal program and allow to pass data collected in a StatPascal vector to Xtremes, thus creating a new active data set.

One should note that the routines described in this section are also available in the command line version of StatPascal. It is possible to read and generate data sets in the format of Xtremes, and the concept of an active data set is also supported by the runtime environment.

We start with an example for generating standard Pareto data under the shape parameter $\alpha = 1$.

```
program Pareto;
const n = 100;
      alpha = 1.0;
var x : vector of real;
begin
    x := paretodata (alpha, n);
    createunivariate (x, 'pareto.dat', 'Description')
end.
```

Here n Pareto data are generated independently by the function *paretodata* and stored in the vector x. The call to *createunivariate* passes the data to Xtremes, that is, the data set stored in x is saved to the file pareto.dat which is then the active one. In addition, a short comment is added. After having run the program, all options of the menu system can be applied to the new data set.

4.4.1 Passing Data from StatPascal to Xtremes

We now provide a more systematic description of the generation of data sets by StatPascal. Four different procedures are provided to pass data collected in a

vector from StatPascal to Xtremes. In the following examples, the data are saved to filename.dat in the working directory. One may create data sets of the following types.

Xtremes Univariate Data: data $x_1, \ldots, x_n$ are collected in a real vector given as argument to the call of the predefined procedure *createunivariate*.

```
var x: vector of real;
...
createunivariate (x, 'filename.dat', 'Description');
```

Instead of a vector, a one–dimensional real array may be given as well.

Xtremes Time Series: in addition to the previous case, a vector containing the times t_i of the observations must be provided.

```
var   x : vector of real;
      t : vector of integer;
...
createtimeseries (t, x, 'filename.dat', 'Description');
```

Xtremes Censored Data: besides a real vector containing the censored data, there is an integer vector with the censoring information.

```
var   z : vector of real;
      delta : vector of integer;
...
createcensored (z, delta, 'filename.dat', 'Description');
```

Xtremes Multivariate Data: the data $x_{i,j}$ are collected in a real matrix. In addition, a string with the column names, separated by '|', must be provided.

```
var x: matrix of real;
    h: string;
...
h := 'Day|Month|...';
createmultivariate (x, h, 'filename.dat', 'Description');
```

Note that a two–dimensional array can be provided instead of a matrix type, because the language supports an implicit type conversion from two–dimensional arrays to matrix types.

As a result of such a procedure you will get an active data set of the type as specified by the command `create...` The Active Data window opens showing the name of your data set and the description provided in the last argument.

4.4.2 Passing Data from Xtremes to StatPascal

Next, let us consider the case where active data are dealt with by StatPascal. The active data set is accessed by means of the following functions:

samplesize	size of the active data set;
dimension	dimension of the active data of type Xtremes Multivariate Data. This function can also be applied to univariate data or a time series, yielding 1 or 2, respectively;
data(i)	$x_{i:n}$ if $x_1, \ldots, x_n$ are Xtremes Univariate Data. Use the function call *data(i,1)* to access the unsorted data;
data(i,j)	$x_{i,j}$ if $(x_{1,1}, x_{1,2}), \ldots, (x_{n,1}, x_{n,2})$ is the active time series. Multivariate data are dealt with in the same way. If a grouped data set is active, then *data(i,1)* returns the cell boundary t_i and *data(i,2)* the frequencies n_i in cell $[t_i, t_{i+1})$. Moreover, censored data are treated like multivariate data with the censored data in the first component, the censoring information in the second and the weights of the Kaplan–Meier estimate in the third one;
columndata(i)	vector with the (unsorted) data in the ith column of the active data set;
rowdata(i)	vector with the data in the ith row of the active data set;
columnname(i)	name of the ith column. This function yields an empty string if not applied to a multivariate data set.

In the following example, we employ StatPascal to add the value 5 to univariate data. Note that the vector structure allows us to deal with data sets of any size.

```
program translation;
var x: vector of real;
begin
    x := columndata (1);
    createunivariate (x + 5, 'demo.dat', '')
end.
```

We used the function call `columndata (1)` to access the unsorted data set.

4.4.3 Temporary Data Sets

One should be aware that the data creation routines described so far result in data being written to disk files and kept in memory until explicitly deleted. In addition,

StatPascal allows the generation of temporary data sets. These data sets are not written to a file and they are deleted as soon as another temporary data set is created.

Within a StatPascal program, temporary data are treated like the usual active data which come from Xtremes. To create a temporary data, specify an empty file name in the call to *create**. For example, the following code generates a temporary data set.

```
var x: vector of real;
...
createunivariate (x, '', '');
```

This feature is useful when a procedure relying on an active data set is employed in a simulation, where one usually creates large amounts of data.

4.5 Predefined Estimators

Predefined estimators for univariate data can be called from StatPascal programs. These procedure calls require a real vector with the data set and reference parameters that are set to the estimated shape (if applicable), location and scale parameters. An additional parameter accepts an error condition. Moreover, one must specify the number of upper extremes when calling estimators in the POT domain.

Suppose the following variables are declared:

```
var k, Result: integer;
    x: vector of real;
    alpha, gamma, mu, sigma: real;
```

With these declarations, one can call the following estimates. The names of these procedures are the same as the corresponding menu options.

POT
```
MLEGP0 (x, k, mu, sigma, Result)
HillGP1 (x, k, alpha, mu, sigma, Result)
MEGP1 (x, k, alpha, mu, sigma, Result)
MomentGP (x, k, gamma, mu, sigma, Result)
DreesPickandsGP (x, k, gamma, mu, sigma, Result)
MLEGP (x, k, gamma, mu, sigma, Result)
```

MAX
```
MLEEV0 (x, mu, sigma, Result)
MLEEV1 (x, alpha, mu, sigma, Result)
MLEEV (x, gamma, mu, sigma, Result)
LRSEEV (x, gamma, mu, sigma, Result)
MDEEV (x, gamma, mu, sigma, Result)
```

```
SUM  MLEGaussian (x, mu, sigma, Result)
     MDEGaussian (x, mu, sigma, Result)
     MHDEGaussian (x, mu, sigma, Result)
     LSEGaussian (x, mu, sigma, Result)
```

The parameter x containing the data can be omitted. In that case, the estimators are applied to the active data set. The parameter *Result* is set to an error status with the following possible values.

0	No error;
1	Negative data in GP1 or EV1 model;
2	Newton–Raphson iteration found no point of zero;
3	k is too small;
4	k is larger than sample size;
5	$x_{n-k+1:n}, x_{n:n}$ have different sign (MomentGP only);
6	Sample size is too small;
8	Internal error.

As a technical example, we simulate the distribution of the MDE(EV) estimator under a Gumbel distribution. The following program applies the estimator to $m = 1000$ data sets with $n = 100$ independently generated values and stores the estimates of the shape parameter in a new active data set. Then, one can use menu options of Xtremes (e.g., *Visualize... Kernel Density* or *Functional Parameters* in the local menu of the *Active Sample* window to analyze the result.

```
program EVSim;
const m = 1000;
      n = 100;
var x, y: vector of real;
    gamma, mu, sigma: real;
    i, result: integer;
begin
    for i := 1 to m do begin
        x := GumbelData (n);
        MDEEV (x, gamma, mu, sigma, result);
        if result = 0 then
            y := combine (y, gamma)
    end;
    createunivariate (y, 'evsim.dat', '')
end.
```

In the above program, we collect the estimates in the vector y, which is passed to *createunivariate* at the end of the program.

4.6 Estimator Programs

An important facility of Xtremes is the estimator dialog box, providing options for the execution of an estimator and for the exploration of the results. With StatPascal, one can implement estimators and attach them to the estimator dialog box. This section shows the required technical details.

Table 4.1 gives an overview of the different domains, distributions and models in the extreme value setting. Estimators of the shape parameter may be implemented by specifying a statistical model after the program name.

Table 4.1: Statistical models.

Domain	Model	Distribution	Shape Parameter
POT	GP0	$W_{0,\mu,\sigma}$	–
	GP1	$W_{1,\alpha,\sigma}$	α
	GP	$W_{\gamma,\mu,\sigma}$	γ
MAX	EV0	$G_{0,\mu,\sigma}$	–
	EV1	$G_{1,\alpha,\sigma}$	α
	EV	$G_{\gamma,\mu,\sigma}$	γ

4.6.1 Implementing Estimators of the Shape Parameter

We start with an artificial example. Enter the following program and save it to a file (e.g., to art.sp).

```
estimator art GP1;
returning alpha;
begin
    alpha := 1.0
end.
```

Next, select *Run* from the button bar in the editor window. Xtremes opens the usual estimator dialog box and displays the results of your estimator (recall that an active data set is required to carry out an estimation procedure). We mention some details.

- The program header defines
  - the name of the estimator (this is art in our example),
  - the model for which the estimator is defined (e.g., *GP1*).

- The identifier (e.g., alpha) after the **returning** symbol defines the name of the estimated parameter and is treated as a global variable of type *real*. The value of this variable will be returned as estimate.

- Changing the active data set while your estimator dialog box is still open does not affect the estimator. You must start the program again to apply the estimator to a new data set.

- Within the GP models, the number of extremes used for the estimation is available by means of the function *extremes*.

- Xtremes provides least squares estimators for the scale and location parameters (e.g., in our artificial example, the least squares estimation is carried out in the Pareto model with fixed shape parameter $\alpha = 1$).

The following example deals with the implementation of the Pickands estimate

$$\hat{\gamma}_k = \log\left(\frac{x_{n-m+1:n} - x_{n-2m+1:n}}{x_{n-2m+1:n} - x_{n-4m+1:n}}\right) / \log 2$$

for $m = [k/4]$ and $4 \le k \le n$ (note that the predefined functions *data*, *samplesize* and *extremes* correspond to x, n and k in this formula):

```
estimator Pickands GP;
returning gamma;

function EstimateGamma (m: integer): real;
    var h1, h2: real;
    begin
        if m < 1 then return 1.0;
        h1 := data (samplesize - m + 1)
                - data (samplesize - 2*m + 1);
        h2 := data (samplesize - 2*m + 1)
                - data (samplesize - 4*m + 1);
        return log (h1 / h2) / log (2)
    end;

begin
    gamma := EstimateGamma (extremes div 4)
end.
```

Note that a **div** b is the notation for $[a/b]$ in StatPascal, where a and b are positive integers. This program is stored in the file pickands.sp in the subdirectory `sp`.

4.6.2 Implementing Estimators of Further Parameters

Within the framework presented above, additional functional parameters (such as the right endpoint) may be returned. Then, the shape parameter must be written as the first identifier after **returning**. For example, write

```
returning alpha, theta;
```

to include a parameter theta. Note that theta will be displayed in the estimator dialog box, and a diagram of the estimated parameters can be plotted in the case of GP models. Yet, parametric curves, such as densities etc., based on theta are not supported.

You may also implement your own estimators for the scale and location parameters. In that case, the word full is added to the model definition, i.e., one must write *FullGP1*, *FullGP*, *FullEV1* or *FullEV* instead of *GP1*, etc. We provide an example for the EV model.

```
estimator art1 FullEV;
returning gamma, mu, sigma;
begin
    gamma := 1.0;
    mu := -0.2;
    sigma := 0.8
end.
```

We mention some details:

- the first parameter after **returning** is the shape parameter. The second and third parameter are treated as location and scale parameters. Additional parameters (like theta introduced above) may be included, and
- predefined estimators are available by means of the functions *hillgp1*, *megp1*, etc.

4.7 StatPascal Runtime Environment

The options of the compiler and the handling of run–time errors are described.

4.7.1 Handling Run–Time Errors

Run–time errors occur during the execution of a program; these are divisions by zero, stack overflows or range check errors when accessing arrays or data sets. If StatPascal encounters such an error, the program terminates and a message showing the type of the error and its address is displayed. When possible, the caret is positioned at the line that caused the error.

Consider the following program (intended to generate a univariate data set):

```
program error;

const n = 100;
var a: array [1..n] of real;
    i: integer;

function invert (x: real): real;
    begin
        return 1 / x
    end;

begin
    for i := 1 to n do
        a [i] := invert (trunc (5 * random));
    createunivariate (n, a, 'test.dat', 'Data set test.dat')
end.
```

This program is likely to crash because the call to `trunc(5 * random)` returns random integer numbers between 0 and 4. As soon as the system performs a division by 0, it displays the error message

```
Division by zero at address ...
```

and starts searching for the run–time error. It then displays

```
Run-time error position found at line 8
```

and places the caret on the line with the statement `return 1/x`.

4.7.2 The Compile Button in the StatPascal Editor

The *Compile* button  is used to compile and/or execute programs. One can choose between the following options.

- *Syntax Check Only.* Compiles the program in the StatPascal window and shows errors that are detected.
- *Compile and Run Program.* Compiles the program and executes it.
- *Compile to File.* Compiles the program and writes the generated code to the specified file. One can append pre–compiled programs to the menu bar of Xtremes by storing them in the *sp*–subdirectory of an Xtremes installation. The programs become availabe the next time Xtremes is started.
- *Find Run–Time Error.* An automatic search for the position of a run–time error position is not possible for pre–compiled programs because they run

independent of an editor window. If a run–time error occurs, one only gets a message showing the type and address of the error in the compiled code. You can load the source code into a StatPascal window and enter the error address to locate its position in the source code.

Note that the *Compile to File*–option allows the distribution of StatPascal programs without disclosing source code.

4.7.3 The Compiler Options Button

The *Compiler Options* button  in the StatPascal editor window opens the dialog box (cf. Fig. 4.1) that provides all options for controlling the compiler. It allows the user to change the memory size and run–time error handling of StatPascal.

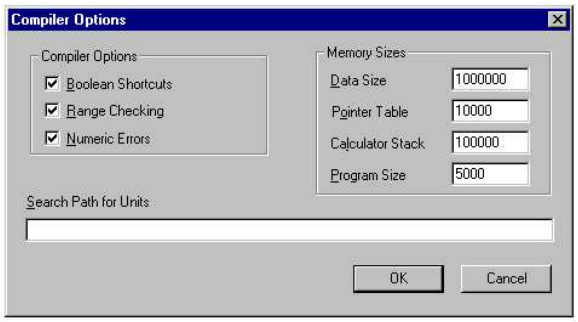


Figure 4.1: The *Compiler Options* box provides options to compile and run a StatPascal program.

Turning off the run–time error checking increases the speed of your program. However, this should only be done for programs which are free of errors, because, otherwise, Xtremes might terminate unexpectedly.

The *Search Path for Units* contains a list of directories where units are searched. Different directories are separated using a semicolon. E.g., the entry `c:\xtremes\sp;d:\myfiles` would instruct the compiler to search within the directories `c:\xtremes\sp` and `d:\myfiles` for units.

The options on the right side of the dialog box control the memory sizes of the abstract stack machine that executes compiled StatPascal programs. It contains of a storage area for program code (*Program Size*) and an area for data (*Data Size*).

The data area must be large enough to hold all objects that are used within a program. The system creates two special areas within the data area whose size is set using *Pointer Table* and *Calculator Stack*. The pointer table keeps track of all pointers and vectors that are created within a program; each of these objects

uses one entry within the table. The calculator stack is used to store intermediate results that occur in calculations.

It is usually safe not to change the default values. One can lower them to spare memory when running a program. Increasing the values may be necessary for programs working with very large data sets. Note that the memory sizes define the number of numerical values (integer or reals) that can be stored; one must multiply these values with eight to obtain the number of bytes required for them.

4.8 Appending StatPascal Programs to the Menu Bar of Xtremes

It is possible to append StatPascal programs to the menu bar of Xtremes. These programs are called like other menu options, it is not necessary to load the source file into the StatPascal editor and to compile it each time the program is started.

To append a program to the menu bar, select the option *Compile to File* in the *Compile* box and store your compiled program under the name `*.out` within the `sp` subdirectory of the working directory. Upon the next start of Xtremes, the system loads all `*.out` files from that directory and creates a separate menu with the names of the programs. The source file is not required to execute the programs, so one can distribute StatPascal programs in binary form.

Chapter 5

Input and Output

This chapter describes the input and output facilities of StatPascal. We start with text–oriented operations in the StatPascal window and a description of the access to text and binary files. The following sections describe the plot routines and facilities to build a graphical user interface.

5.1 The StatPascal Window

The procedures *write*, *writeln* and *read* of the Pascal language are provided to perform input and output operations in the StatPascal window. The are several predefined procedures which allow the control of the ouput. The Page (see Section 7.119) procedure is used to clear the contents of the StatPascal window, while TextColor (see Section 7.182) and TextBackground (see Section 7.181) set the color of subsequent output operations.

The output position can be adjusted with the GotoXY (see Section 7.75) procedure.

5.2 File Operations

Two different types of files are provided by StatPascal. Text files are used to store information in human readible form, while binary files are employed to store values using the internal representation of StatPascal.

5.2.1 Text Files

The predefined data type *text* is used to address text files. The following example shows the creation of a text file:

```
var f: text;
```

```
begin
    assign (f, 'demo.txt');
    rewrite (f);
    writeln (f, 'Output to demo.txt');
    close (f)
end.
```

After running this program, the working directory contains a file `demo.txt` with the text line `Output to demo.txt`.

You can write all values that are legal as arguments of *write* or *writeln* in its standard form, i.e. values of the types *integer, char, boolean, real* and *string*. It is not possible to output other types or data structures.

5.2.2 Binary files

Binary files store the internal representation of objects used by StatPascal. This format is not readible by humans; such files may only be accessed from an StatPascal-program. The main advantage is the possibility to store objects of any type.

Users familiar with Turbo Pascal should note that binary files are not typed in StatPascal. It is therefore possible to store objects of different data types within a file. Turbo Pascal provides the predefined procedures *BlockRead* and *BlockWrite* for that purpose.

The following example shows the storage of a real array to a binary file.

```
var a: array [1..n] of real;
    f: file;
begin
    assign (f, 'demo.bin');
    rewrite (f);
    write (f, a);
    close (f)
end.
```

Then, to read the array back into a program, the following code can be used.

```
var a: array [1..n] of real;
    f: file;
begin
    assign (f, 'demo.bin');
    reset (f);
    read (f, a);
    close (f)
end.
```

5.3 Plots

StatPascal allows the user to open an Xtremes plot window and to display curves and scatterplots in it. These windows and curves exactly act like the ones available from the menu system.

5.3.1 Univariate Curves

Xtremes provides a predefined function Plot (see Section 7.126) which is utilized to plot univariate curves. The function requires two vectors containing the points x_i and values $f(x_i)$, the destination window and a description of the curve. A linear interpolation of the given points is displayed.

For example, the following program plots a Gaussian density in two Xtremes plot windows.

```
program gaussplot;
const n = 100;
var x, y: vector of real;
begin
    x := realvector (-3, 3, n);
    y := gaussiandensity (x);
    plot (x, y, 'Density 1', 'Gaussian density');
    plot (x, y, 'Density 2', 'Gaussian density')
end.
```

Two Xtremes plot windows (*Density 1* and *Density 2*) are opened by calls to *plot*. The curves are displayed as solid black lines. One can change their appearance using different plot options (see Section 5.3.5).

Please note that the points provided in the vector `x` do not need to be in increasing order. Thus, it is possible to plot any line segment.

5.3.2 Scatterplots

The ScatterPlot (see Section 7.149) procedure is similar to the Plot (see Section 7.126) procedure. The routine requires three parameters: two arrays defining the points and the name of the scatterplot window. In the above example, the call to plot must be replaced by `scatterplot (x, y, 'Scatterplot')` to obtain a scatterplot of the points $(x_1, y_1), \ldots, (x_n, y_n)$.

5.3.3 Contour and Surface Plots

StatPascal also performs contour (see Section 7.127) and surface (see Section 7.128) plots of a bivariate function f. Given a rectangle $[x_0, x_1] \times [y_0, y_1]$, one has to define a real matrix or a two–dimensional array `v: array [1..n, 1..m] of real`, and store the values $f(x_0 + (i-1)/(n-1)(x_1 - x_0), y_0 + (j-1)/(m-1)(y_1 - y_0))$

in the indices `v [i, j]`. Then, the calls to `PlotContour (x0, x1, y0, y1, v)` and `PlotSurface (x0, x1, y0, y1, v)` produce a contour plot or surface plot of f.

The following example displays a contour plot and a surface plot of a bivariate Gaussian density.

```
program plotdemo;
var y: array [-30..30, -30..30] of real;
    i, j: integer;
begin
    for i := -30 to 30 do
        for j := -30 to 30 do
            y [i, j] := gaussiandensity (i / 10) *
                        gaussiandensity (j / 10);
    plotcontour (-3, 3, -3, 3, y);
    plotsurface (-3, 3, -3, 3, y)
end.
```

5.3.4 Polygons

5.3.5 Overview Of Advanced Plot Options

The appearance of plot windows and curves can be controlled from a StatPascal program by calling predefined procedures. The user can select a coordinate system, attach labels to a plot or change the color and line style of curves.

Coordinate System: the coordinate system of a window is set by calling the procedure SetCoordinates (see Section 7.152) which changes the coordinates in the window specified by the first parameter. One can control the ticks and markers displayed on the axes by calling SetTicks (see Section 7.161) and SetMarkers (see Section 7.157).

Labels: use the procedure SetLabel (see Section 7.153) to place a text label into a window, whereby the the alignment of the label as well as its orientation can be controlled. You can also place a label at the edges of the window.

Curves: one can select the color of a curve by calling SetColor (see Section 7.150). The plot style (i.e. the appearance of the curve, like displaying a linear interpolation or just the supporting points) is controlled by SetPlotStyle (see Section 7.159). Use SetLineStyle (see Section 7.156) to activate a predefined line style (like solid or dashed) or define your own one by using SetLineOption (see Section 7.155).

We provide an example for the use of the advanced plot options. The following StatPascal program displays a dotted Pareto density and a dashed Frechet density. The ticks and markers of the coordinate system are selected by the program, and a title is added to the plot. Note that because of the implicit type conversion from arrays to vectors, a one–dimensional array can be given as argument to Plot (see Section 7.126).

```
program PlotDemo;
const win = 'Density';
var x, y: array [40..80] of real;
    i: integer;
begin
    for i := 40 to 80 do begin
        x [i] := i / 10;
        y [i] := paretodensity (1, x [i])
    end;
    SetLineStyle (DottedLine);
    Plot (x, y, win, 'Pareto');
    for i := 40 to 80 do
        y [i] := frechetdensity (1, x [i]);
    SetLineStyle (DashedLine);
    Plot (x, y, win, 'Frechet');
    SetTicks (win, 4, 8, 0.5, 0, 0.1, 0.05);
    SetMarkers (win, 4, 8, 1, 0, 0.1, 0.1);
    SetLabel (win, VerticalPositionTop or
            HorizontalPositionCentered, 0, 0, 'Densities')
end.
```

5.4 Graphical User Interface

StatPascal provides three predefined routines for the construction of graphical user interfaces. The call to MessageBox (see Section 7.106) opens a dialog box with the given string argument and an OK–button.

MenuBox (see Section 7.105) is used to construct menus. It expects two strings yielding the title of the menu and the single entries, separated by |. The number of the selected menu option is returned (counting starts with 1), a zero indicates that the menu was cancelled by the user.

The DialogBox (see Section 7.33) function allows the contruction of dialog boxes with up to 10 real input values. Its first two arguments are similar to the *MenuBox* function. They provide a title and list with the single entries. The third argument is a real vector with the default values of the input fields. A vector with the input values is returned; it is empty if the dialog is cancelled by the user.

Chapter 6

Syntax of StatPascal

This chapter provides a concise definition of the StatPascal syntax, starting with the lexical elements. We utilize syntax diagrams because they are readible easily. Lower and upper case letters can be used. StatPascal is not case-sensitive, so `WHILE`, `while` or `WhilE` mean the same.

6.1 Lexical Elements

We start with the lexical elements that are handled by the scanner.

6.1.1 Special Symbols and Reserved Words

A StatPascal program is built using the following symbols.

- Letters A to Z, a to z.
- Digits 0 to 9.
- Whitespaces (blanks, tabulators and line feeds).
- The following special symbols.

+	-	*	/	**	!	:=
;	:	=	<	>	<>	<=
>=	(	)	[	]	..	%
(*	*)	,	.	^	”	’
{	}					

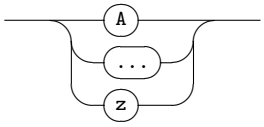
StatPascal reserves the following words which cannot be used as identifiers.

and array begin case component div do else end estimator for forward function goto if implementation in interface label matrix mod new nil not of or procedure program record repeat return returning set then to unit until uses var vector while xor

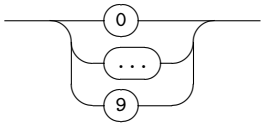
6.1.2 Identifiers

An identifier starts with a letter followed by letters or digits, it ends with a white-space or one of the specials symbols listed above. Identifiers may be of arbitrary length, but only the first 30 characters are significant. This means that two identifiers must be different within the first thirty characters.

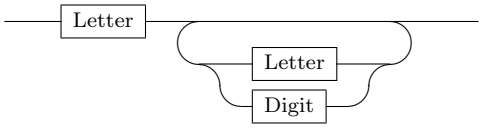
Letter



Digit



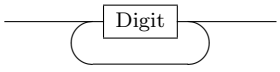
Identifier



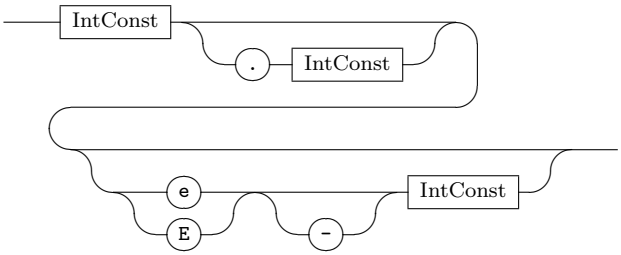
6.1.3 Numerical Constants

Integer constants are written as a sequence of decimal digits as shown in the following syntax graph (with digit representing one of the symbols 0, ..., 9.) Real numbers contain an exponent or a fractional part.

IntConst



UnsignedNumber



Negative numbers are produced within the context of expressions.

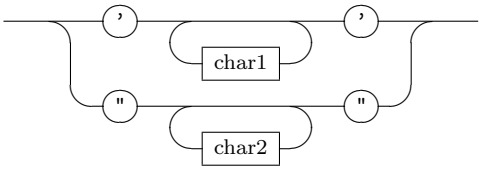
The following example shows integer and real numbers.

integer	4631	71	42	
real	3.1415	2.71e5	3e-4	1.0

6.1.4 Character and String Constants

Character constants are included in single apostrophes, e.g. 'A'. String constants are included in " or '. They may contain any character except the chosen quotation marks, line breaks are not allowed within them. The following syntax applies to string constants.

StringConst



Here *char1* denotes any character except ", while *char2* denotes any character except '. Examples of string constants are:

'pareto.dat'
"Data set containing last year's maxima"
'He said: "Hi there."'
""

Note that a single character included in primes ' denotes a *char* constant, while multiple characters or a single character surrounded by " define a constant of the *string* type.

6.1.5 Comments

Comments may be inserted between identifiers or special symbols. Any text within a comment is ignored. A comment starts with **(*** or **{** and ends with ***)** or **}**. The TEX–format of comments is also supported. Any text following a %-sign is regarded

as a comment until the end of the line where the %-sign occured. The following example shows the use of comments.

```
xplprogram gaussiandensity;
(* This program displays a table
   of Gaussian densities *)
var i: integer;
begin
    for i := -30 to 30 do
        writeln (i / 10, ' ', gaussiandensity (i))
end. % End of program
```

A comment is treated as a white space.

6.2 Blocks

A block consists of four different parts followed by a statement. The optional parts define labels, constants, types, variables and declare functions and procedures. All identifiers declared within a block are local to that block. One obtains nested blocks by declaring functions and procedures. Any identifier declared within an outer block is accessible in an inner block. Such identifiers may be redeclared in the inner block; in this case the original meaning is hidden and restored at the end of the inner block.

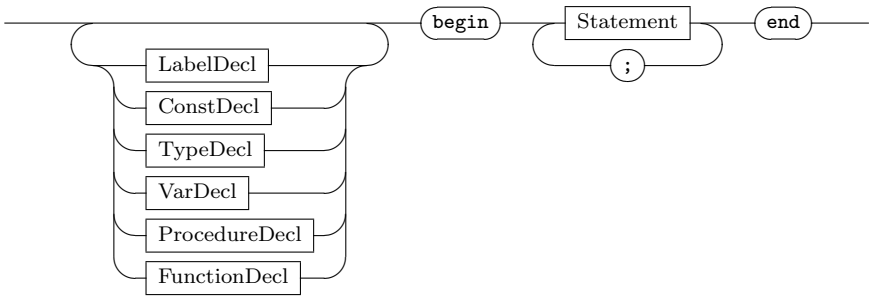
The first optional part of a block starts with the reserved word **label** followed by label declarations. The others are used to declare constands, data types and variables. They start with the reserved words **const**, **type** and **var**.

Procedures and functions may be defined in any order. However, if a function or procedure calls another one it is necessary that the called routine is defined first.

The strict ordering of the different parts of a block required by Pascal is not necessary within StatPascal. In addition, each part may be given more than once. This modification allows the inclusion of routines depending on their own variable or type declarations by means of include directives.

The last part of the block is mandatory: A sequence of statements bracketed with **begin** and **end** must be used. A block takes the following general form.

Block



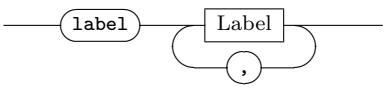
The next example shows all parts of a block. Note that the procedure defines a nested block with a local identifier.

```
label lbl;

const n = -30;
      m = 30;

type  field = array [n..m] of real;

var   a: field;

procedure fill (var c: field);
    var i: integer;
    begin
        for i := n to m do c[i] := GaussianDF (i / 10)
    end;

begin
    goto lbl;
    writeln ('Not executed');
lbl:
    writeln ('Filling array');
    fill (a)
end
```

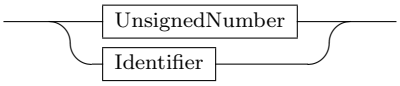
6.3 Labels

A label is used as a target of a **goto** instruction, which unconditionally transfers program execution to the statement after the label. A label may be an identifier or an integer number.

LabelDecl



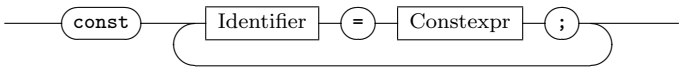
Label



6.4 Constants

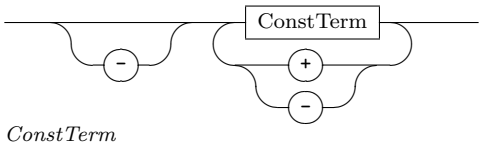
A constant declaration defines an identifier representing a constant value within the block it appears in. The constant declaration starts with the reserved word **const** followed by a list of identifiers set equal to constant expressions. Each definition ends with a semicolon. The general form of the constant declaration is

ConstDecl

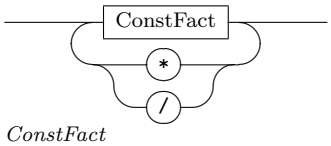


A constant expression is build by numbers or by constants defined previously using the operators $+$, $-$, $*$ and $/$. Its syntax is given by

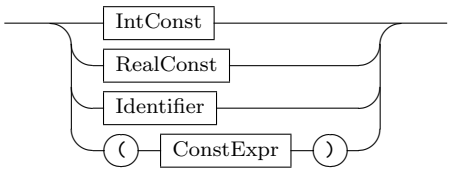
ConstExpr



ConstTerm



ConstFact



The following example demonstrates the declarations of constants:

```
const a = 3.1415;
      b = 7;
      c = 2.5 * (a + b);
      s = 'String constant';
      d = 'A';
```

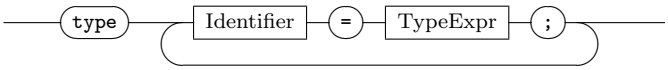
6.5 Types

StatPascal is a typed language, i.e. each object belongs to a data type that defines the set of possible values of an object. The language provides predefined data types (*boolean*, *char*, *integer*, *real*, *string*, *text*, *file*, *realvector*, *real2* and *real3*) and type constructors to introduce new data types.

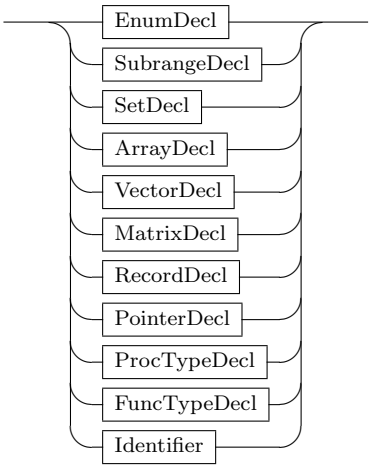
6.5.1 Type Declarations

StatPascal permits the declaration of new data types. A name is assigned to a data type in a type declaration taking the form

TypeDecl



TypeExpr



with *typeexpr* representing a predefined or previously declared data type or a data type created using one of the type constructors. A pointer to a data type that appears later in the type declaration may be declared.

The following example shows some type declarations (note especially the recursive declaration of the types *nptr* and *node*).

```
type field  = array [1..10] of real;
     rec    = record
                a, b: field;
                s: string
              end;
     colset = set of (red, green, blue, brown, black);

     nptr   = ^node;
     node   = record
                contents: rec;
                next: nptr
              end;

     realf  = function (real): real;
```

```
realp  = procedure (real);
```

6.5.2 Simple Types

6.5.2.1 Ordinal Types

Ordinal types are ordered data types with an injective mapping from the set of their values to a subset of the natural numbers.

Boolean *boolean* is a predefined data type representing logical values. It contains the values *true* and *false* which are available as predefined identifiers. The relation *false* < *true* holds.

Char The data type *char* represents single characters.

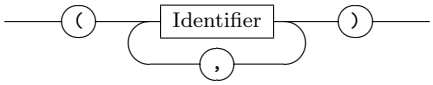
Integer *integer* contains the integer numbers which may be represented in a memory word of the machine. Its length is therefore machine dependent.

Enumerations New ordinal data types are declared by enumerating all their values, e.g.

```
type colors = (red, green, blue, brown, black)
```

When *ord* is applied to a value of an enumerated type, it returns the position of the value in the list (starting with zero).

EnumDecl



Subranges Subranges of enumerated types are defined by specifying the lower and upper bounds of the subrange type, e.g.

```
type index = 1..15;
     color = red..blue;
```

SubrangeDecl

6.5.2.2 Reals

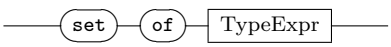
The data type *real* contains floating point numbers using the internal representation for double precision values.

6.5.3 Structured Data Types

Various type constructors are provided to build structured data types based on scalar types.

6.5.3.1 Sets

Sets of enumerated types are introduced using the set constructor
SetDecl

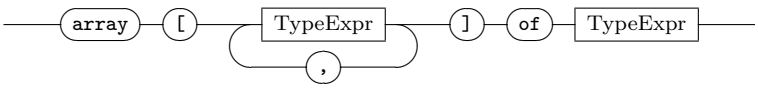


where *TypeExpr* must yield an enumerated data type with ordinal values between 0 and 255. Sets may be united or intersected using the operators + and *, the operator - calculates the difference set. Sets are constructed by including expressions in square brackets, e.g. `[1, 3, 7]` or `[red, blue]`. The empty set is denoted by `[]`.

The boolean operator **in** is provided to check if a value is a member of a set, e.g. the expression `red in [red, blue]` evaluates to true.

6.5.3.2 Arrays

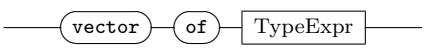
Arrays are defined using the array constructor
ArrayDecl



The type expressions within the square brackets define the index types used to access the components of the array. These types must be ordinal.

6.5.3.3 Vectors

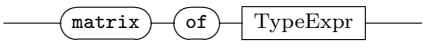
A vector type is obtained by the type constructor
VectorDecl



Within expressions, integer vectors are constructed by listing the first and last value, seperated by two dots, e.g. `3..10`.

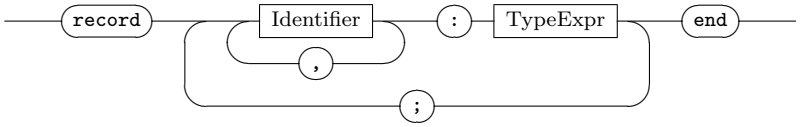
6.5.3.4 Matrices

Matrices are defined by the matrix construction
MatrixDecl



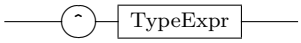
6.5.3.5 Records

Records are declared using the record constructor
RecordDecl



6.5.3.6 Pointers

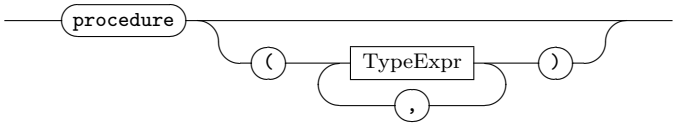
Pointer types are declared utilizing the constructor
PointerDecl



6.5.3.7 Procedural and functional types

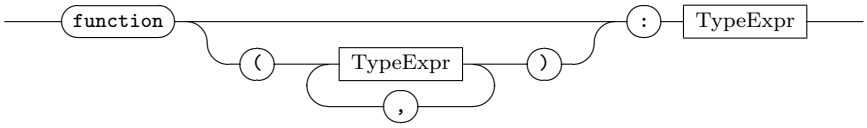
Procedural and functional types are constructed by means of the type constructors

ProcTypeDecl



and

FuncTypeDecl



6.5.4 Compatible Types

Two data objects are of the same data type if they were declared using the same type constructor or if they are objects of the same named data type. Reference parameters of a subroutine expect actual parameters of the same type as the formal parameter.

If a value is to be assigned to another one, or if two values are combined using a binary operator, type compatibility is required. Assignment of a value having data type v to a variable of type u is possible in the following cases.

1. u and v are the same types.

2. u and/or v are subranges of the same type. If u is a subrange type, a range check is performed.

3. v is a parameterless function returning type w. If w and u are compatible, the function is called, and the return value of the function is assigned to the variable.

4. u and v are functional or procedural types with the same number and types of parameters and the same return type.

5. v is an integer type or a subrange thereof, and u is real.

6. u and v are vectors of the same type, or u is a real and v an integer vector.

7. u is a vector of a type w, and v has type w or is an array of type w.

8. u is matrix of a type w, and v is a two–dimensional array of type w.

If the operands of a binary operator belong to different types u and v, the following rules apply.

1. u and v are subranges of the same type w: The result is of the base type w.

2. u and/or v are parameterless functions: The functions are called, and the rules are applied to their return types.

3. u is real, and v is integer or a subrange thereof. Then v is converted to real, and the result is real.

4. u and v are integer or real vectors. If u is a real vector and v an integer vector, then v is converted to a real vector.

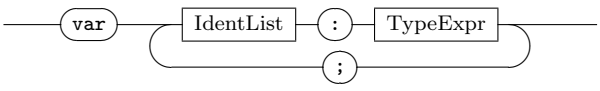
5. v is an integer or a real vector, and u is an integer or real value or array. Then, u is converted to a vector and the above rule applied.

6.6 Variables

6.6.1 Variable Declarations

Variables must be declared in the declaration part of the block where they are used in or in the declaration part of a surrounding block. A variable declaration starts with the reserved word **var** followed by a list of identifiers and the data type associated with them. It has the form

VarDecl



with *IdentList* representing one or more identifiers seperated by commas. Note that *TypeExpr* may be any of the data types described in section 6.5. Variables are local to the block, where they are declared.

The following example shows a variable declaration.

```
var a: field;
    c, d: array [1..5] of real;
    d: (red, green, blue, brown, black);
```

6.6.2 Accessing Variables

A variable access denotes a (part of a) variable. It can be used in expressions as well as in the target of an assignment. A variable access starts with the name of the variable; arrays, records and pointer types provide further possibilities that are shown below.

6.6.2.1 Arrays

An index given in square brackets is used to select array components. If a multidimensional array is accessed, one may give more than one index inside the brackets.

Consider the following variable declarations.

```
var a: array [1..10] of real;
    b: array [char, boolean, 7..15] of (red, green, blue);
```

Then the following variable accesses are legal.

```
a               % entire array
a[7]            % one component
b['A'][false]   % is the same as the next line
b['A', false]   % which is an array of the enumeration
```

Note that vector indices are treated syntactically like array indices; yet, the do not yield l-values, i.e. it is not possible to assign values to a part of a vector.

6.6.2.2 Records

A component of a record is accessed by means of the point operator. An example is given.

```
var i: record
         re, im: real
       end;
begin
    i.re := 0.0;
    i.im := 1.0
end.
```

6.6.2.3 Pointers

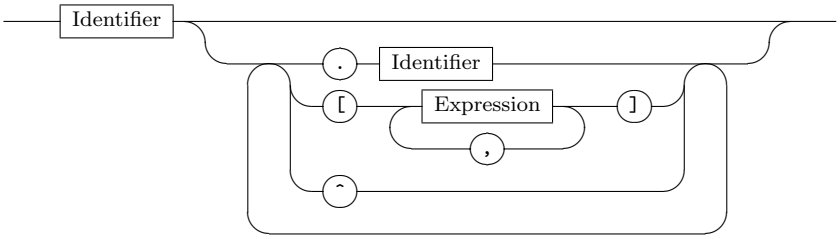
A pointer is dereferenced utilizing the ^–operator. The following example illustrates the usage of a pointer.

```
var a: ^integer;
begin
    new (a);
    a^ := 5
end.
```

6.6.2.4 Syntax

The following syntax diagram defines a variable access.

VarAccess



6.7 Expressions

The usual notation for expressions is supported. Constants, variables and function calls may be combined using the operators shown in the following table.

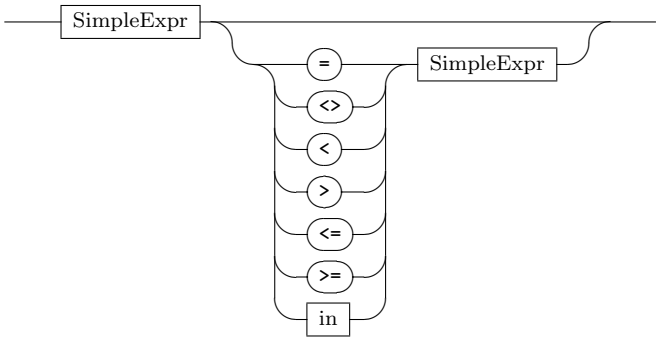
^	.	[]				
**	not					
*	/	div	mod	and		
+	-	or	xor			
=	<	>	<=	>=	<>	in

These operators have the same meaning as in Pascal. Note that ** evaluates powers. Operators are presented in descending order. The pointer dereference operator has the highest priority, while the relational operators have the lowest. Operators belonging to the same group are evaluated from left to right. Brackets are used to change the order of evaluation.

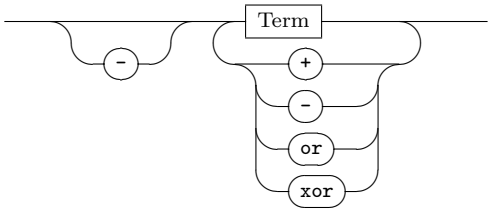
6.7.1 Syntax

The following diagrams define the syntax of expressions.

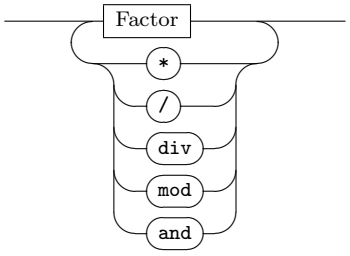
Expression



SimpleExpr



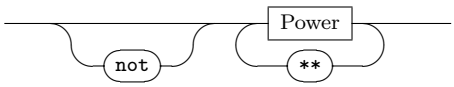
Term



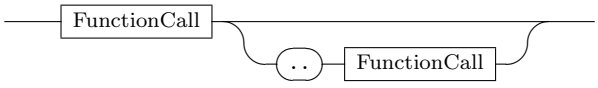
These productions are the same as in the Pascal language. Next, we introduce the power operator and the constructor for *integer* vectors.

Note that the vector constructor has a high priority to ease the use of vectors that are produced by it in expressions. Exercise caution with expressions like `1..a - 1` which generates the vector $0, \ldots, a-1$. One must utilize brackets as in the expression `1..(a - 1)` to obtain the vector $1, \ldots, a-1$.

Factor

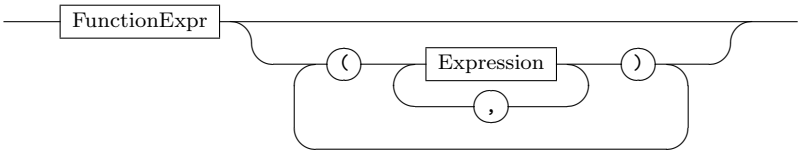


Power



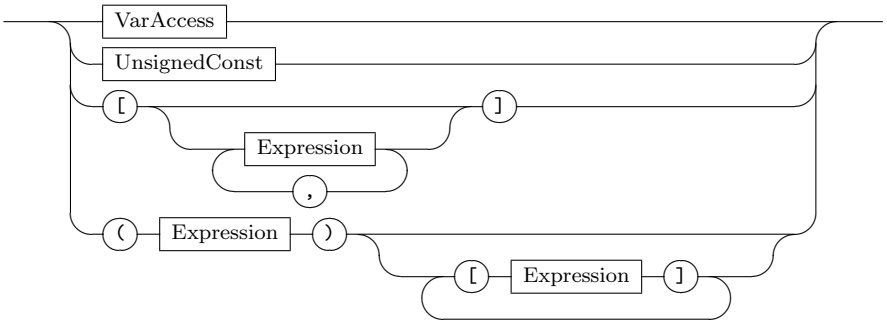
The production for a function call is seperated from the production *FunctionExpr* to introduce functional and procedural variables. Moreover, a function may return a functional or procedural value, so successive calls are allowed (see section 6.8.2 for an example).

FunctionCall

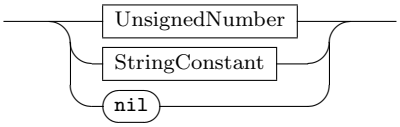


The production *FunctionExpr* is similiar to its Pascal equivalent. Vector expressions given in parantheses may be followed by optional index operations.

FunctionExpr



UnsignedConst



6.7.2 Operators

6.7.2.1 Arithmetic operators

Note that the operators **div**, and **mod** can merely be applied to *integer* operands. The logical operators **and**, **or**, **xor** and **not** perform a bitwise operation when applied to integers.

Dividing two integers using the / operator produces a result of type *real*, **div** and **mod** (giving the whole part of the ratio and the remainder) can be used when an *integer* result is required.

6.7.2.2 Logical operators

The logical operators **and**, **or**, **xor** (exclusive or) and **not** perform the usual logical operations when applied to *boolean* values.

6.7.2.3 Relational operators

The relational operators `<`, `>`, `<=`, `>=`, `=` and `<>` may be applied to values of all ordinal types as well as *real* and vectors thereof. They return *true* (*boolean*) if the condition is met and *false* if not.

6.7.2.4 String operators

Strings are represented by character vectors. The `+` operator concatenates two arguments of the predefined *string* type, and the relational operators perform an alphabetical comparison rather than a componentwise one.

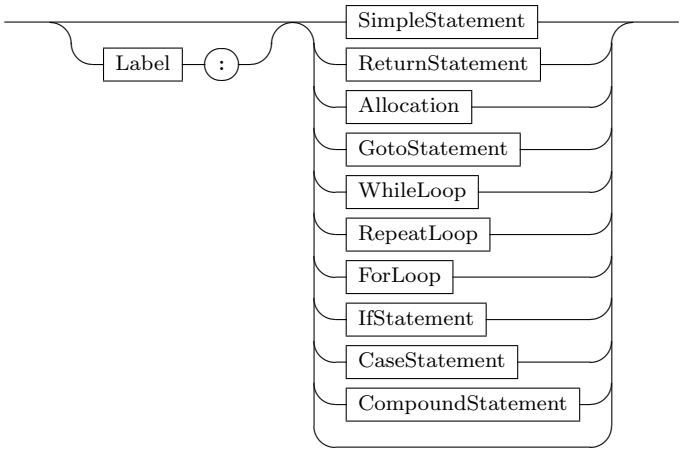
6.7.3 Function Calls

A function call requires a list of actual parameters, corresponding to the formal parameters specified in the declaration of the function. Parameters that are passed by reference require an l-value of the same type, while a compatible type (see section 6.5.4) suffices for parameters passed by value.

6.8 Statements

Statements define the actions taking place when a program is run. A statement is defined by the following syntax diagram.

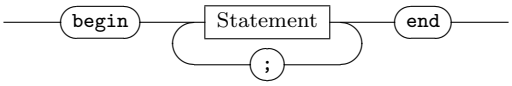
Statement



6.8.1 Compound Statements

Two or more statements may be grouped together seperated by semicolons and surrounded by **begin** and **end**. Such a compound statement may always be used when a single statement is allowed.

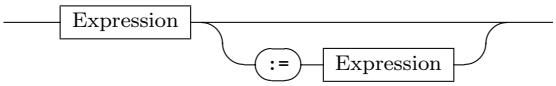
CompoundStatement



6.8.2 Simple Statements

Simple statements perform assigments and procedure calls.

SimpleStatement



An expression is assigned to a variable using the assignment operator `:=`. The types on both sides of the assignment must fulfill the type compatibility as described in section 6.5.4, and the expression used on the left side of the assignment must denote an l-value.

In the most simple case, a procedure is called by using its name as a statement. Formal parameters used in the declaration of the procedure require a list of actual parameters of the same or a compatible type.

Internally, a procedure is treated as a function with a special empty return type *void*. Thus, a procedure call is represented by an expression with the type *void*. The following construction is therefore possible.

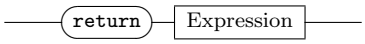
```
type proc = procedure (integer);

procedure proc1 (a: integer);
    begin
        write (2 * a)
    end;

procedure proc2 (a: integer);
    begin
        write (a / 2)
    end;

function f (i: integer): proc;
    begin
        if i = 1 then f := proc1
        else f := proc2
```

```
    end;

begin
    f (1)(2)        (* 4 *)
end
```

6.8.3 Return Statement

A **return** statement is used to terminate a function and return a value to the caller. Its syntax is defined by

ReturnStatement

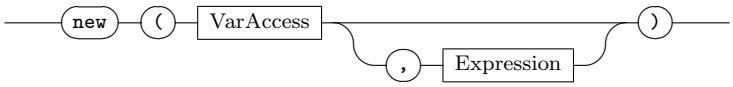


The type of the expression must be compatible to the type of the function. The result of a function may also be defined by an assignment to the function name.

6.8.4 Memory Allocation

Memory is allocated for a pointer type using the **new**–statement.

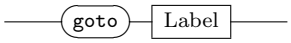
Allocation



6.8.5 Goto Statement

A **goto** statements continues the exeution of the program at the first statement after the specified label. It is not allowed to jump out of a subroutine or into the body of a **for**-loop.

GotoStatement



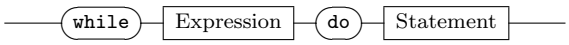
6.8.6 Iterations

Three iterations allow the controlled repetition of a statement.

6.8.6.1 while-Loop

The **while**-loop has the form

WhileLoop

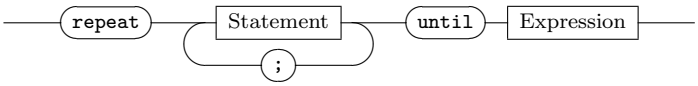


The statement is repeated as long as the expression is true. Since the expression is evaluated before the first execution of the statement, it is possible that the statement is skipped.

6.8.6.2 repeat-Loop

The **repeat**-loop has the form

RepeatLoop

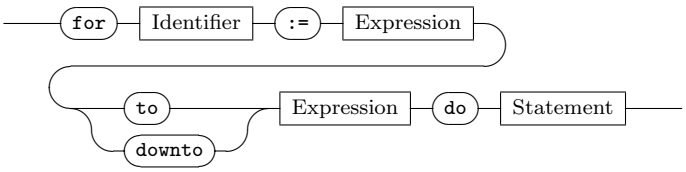


The statements are repeated until the expression is true. Note that the statements are executed at least once since the expression is tested after the execution of the statements.

6.8.6.3 for-Loop

The third loop available has the form

ForLoop



At the beginning of the loop both expressions are evaluated. They must yield values of compatible enumerated data types. The statement is repeated with increasing (decreasing) values of identifier running from the first to the second ordinal value.

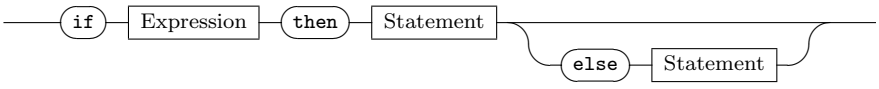
6.8.7 Selections

One of several statements may be executed depending on certain conditions. Stat-Pascal provides two different statements to perform selections.

6.8.7.1 if-Statement

The **if**-statement allows the execution of a statement depending on a single condition. It has the form

IfStatement

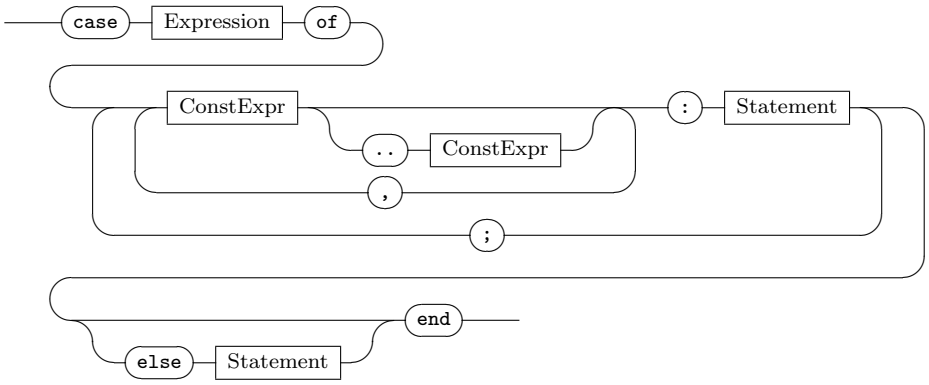


The statement after **then** is executed if the expression is *true*. If the expression is *false* and an **else** part is provided, then the statement following the **else** is executed, otherwise control passes to the next statement. An **else** belongs to the most recently used **if** in nested **if**–statements.

6.8.7.2 case-Statement

The **case**-statement consists of an expression (the selector) and a list of branches. Each branch starts with a list of constants and ends with a statement.

CaseStatement



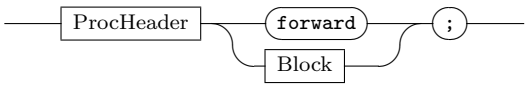
If the selector is equal to a label, the corresponding statement is executed. The **case**–statement is terminated after the first match, other branches are not tested. If the selector does not match a constant, control passes to the next statement, or the optional **else** part is executed. Selector and constants may be of the types *boolean*, *char* or *integer*.

6.9 Procedures and Functions

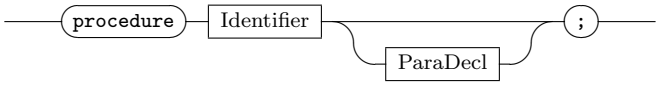
Procedures and functions may be defined in an StatPascal program. Both types of subroutines have an optional parameter list that may be used to pass arguments.

A procedure consists of a header defining its name and parameters followed by a block or the reserved word **forward**. It ends with a semicolon.

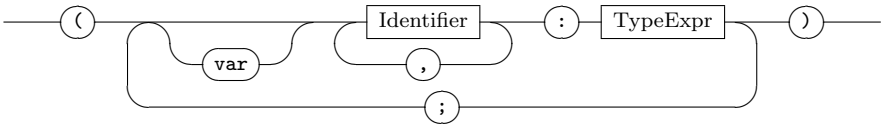
ProcDecl



ProcHeader



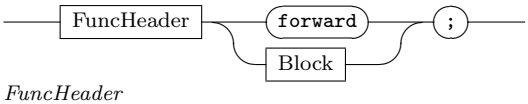
ParaDecl



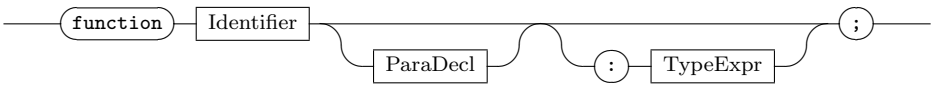
Formal parameters preceeded by a **var** keyword are reference parameters.

A function has a similiar structure; it starts with the reserved word **function** and provides an additional return type.

FuncDecl



FuncHeader



A function may contain a **return** statement specifying its result. Control is passed to the caller immediately after the execution of a **return** statement. Functions and procedures may call themselves recursively.

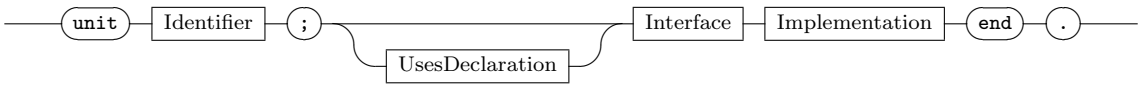
The return type of a function must only be omitted when a previous forward declaration is actually defined. In that case, one must also not repeat the parameter list. The following example shows the forward declaration and definition of a procedure and a function.

```
procedure p (n: integer); forward;
function f (x: real): real; forward;

procedure p;
    begin
    ...
    end;

function f;
    begin
    ...
    end;
```

6.10 Units

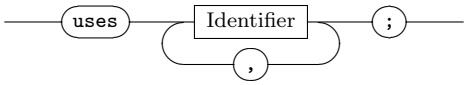
A unit starts with the reserved word **unit**. It consists of an interface section with the public declarations and an implementation section with private declarations and the definition of the subroutines declared in the interface section. A unit thus has the form

Unit



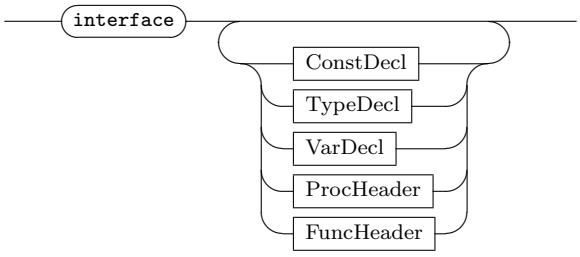
whereby the *UsesDeclaration* list further units required by the actual one.

UsesDeclaration



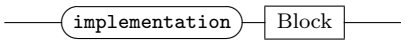
The interface section is defined by

Interface



Finally, the implementation section is given.

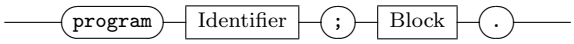
Implementation



6.11 Programs

A program consists of an optional program header defining the name of the program and a program block followed by a point. It has the form

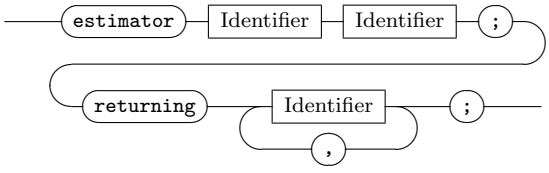
Program



The given name serves only as a comment, it is not used within Xtremes.

The implementation of an estimator requires an estimator header followed by a program block and a point. The estimator header has the form

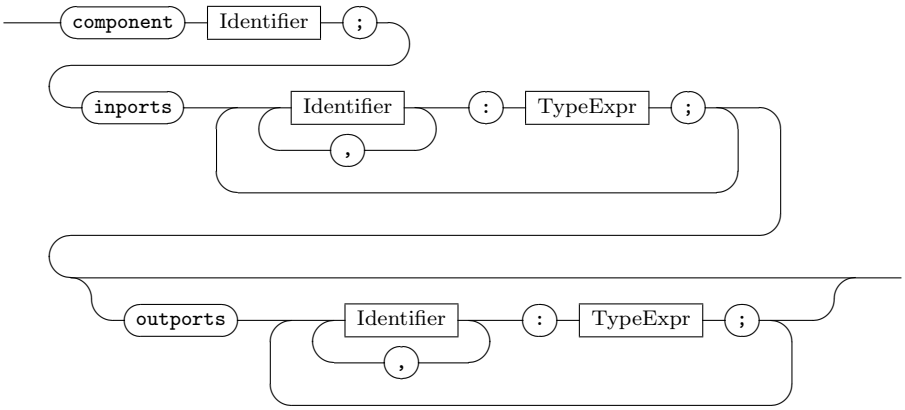
Estimator



with the second *Identifier* being one of the words *gp*, *gp1*, *ev*, *ev1*, *fullgp*, *fullgp1*, *fullev* or *fullev1*.
The names after **returning** define the parameters that are estimated. They are used as global variables of the type *real* within the program, the values assigned to them at the end of the program will be returned to Xtremes.

Finally, a node is defined by

Node



Chapter 7

The StatPascal Library Functions

This chapter provides an alphabetical list of all predefined functions and procedures available in StatPascal and UserFormula. The entry *Availability* shows if a function is implemented in UserFormula and/or StatPascal. Functions available in standard Pascal or other programming languages are only explained briefly.

One should note that the densities, dfs and qfs of most distributions implemented in XTREMES are available in their standard form; e.g. the function call `paretoqf(2,x)` evaluates the qf of the Pareto distribution with shape parameter 2 at the point x. Location and scale parameter must be given explicitly, e.g., by writing `4 + 5 * paretoqf(2,x)` to utilize a scale parameter 5 and a location parameter 4.

Each function and procedure contains entries showing its declaration, a description and its availability. Routines marked with UFO can be applied within the UserFormula facility, routines marked with StatPascal are available in StatPascal programs. Vectorized versions of a function cannot be used in UserFormula.

7.1 abs

```
function abs (x: real): real;
function abs (n: integer): integer;
function abs (x: realvector): realvector;
function abs (x: integervector): integervector;
```

returns the absolute value of the argument.

Available in UFO, StatPascal

```
var x: real;
begin
  x := -2;
  writeln (abs (x))
end.
```

7.2 All

```
function All (b: vector of boolean): boolean
```

returns true iff all components of the boolean vector b are true.

Available in StatPascal

```
var x: vector of real;
begin
  x := GaussianData (100);
  if All (x < 3) then
    writeln ('All values are below 3')
end.
```

7.3 arccos

```
function arccos (x: real): real;
function arccos (x: realvector): realvector
```

returns arcus cosine of x.

Available in UFO, StatPascal

```
var x: real;
begin
  x := 0.5;
  writeln (arccos (x))
end.
```

7.4 arcsin

```
function arcsin (x: real): real;
function arcsin (x: realvector): realvector
```

returns arcus sine of x.

Available in UFO, StatPascal

```
var x: real;
begin
  x := 0.5;
  writeln (arcsin (x))
end.
```

7.5 arctan

```
function arctan (x: real): real;
function arctan (x: realvector): realvector
```

returns arcus tangent of x.

Available in UFO, StatPascal

```
var x: real;
begin
  x := 1.0;
  writeln ('Pi = ', 4 * arctan (x))
end.
```

7.6 BetaData

```
function BetaData (alpha: real): real
function BetaData (alpha: real; n: integer): realvector
```

generates standard Beta (GP 2) data under the shape parameter *alpha*. The vectorized version generates a real vector with n independent realizations.

Available in UFO, StatPascal

```
const alpha = 1.0;
var x: vector of real;
begin
  x := BetaData (alpha, 100)
end.
```

7.7 BetaDensity

```
function BetaDensity (alpha, x: real): real
```

returns the density of the Beta (GP 2) distribution with shape parameter *alpha*.

Available in UFO, StatPascal

```
const alpha = 1.0;
var x: real;
begin
  x := 1.0;
  writeln (BetaDensity (alpha, x))
end.
```

7.8 BetaDF

```
function BetaDF (alpha, x: real): real
```

returns the df of the Beta (GP 2) distribution with shape parameter *alpha*.

Available in UFO, StatPascal

```
const alpha = 1.0;
var x: real;
begin
  x := 1.0;
  writeln (BetaDF (alpha, x))
end.
```

7.9 BetaQF

```
function BetaQF (alpha, x: real): real
```

returns the qf of the Beta (GP 2) distribution with shape parameter *alpha*.

Available in UFO, StatPascal

```
const alpha = 1.0;
var x: real;
begin
  x := 1.0;
  writeln (BetaQF (alpha, x))
end.
```

7.10 BeginMultivariate

```
procedure BeginMultivariate (size, dimension: integer)
```

The procedure *BeginMultivariate* initializes a new multivariate data set. The parameters *dimension* and *size* define the size of the data set. Use SetColumn (see Section 7.151) and EndMultivariate (see Section 7.36) to insert the data points and to provide a filename. An example is provided under SetColumn (see Section 7.151).

Available in StatPascal

7.11 BinomialData

```
function BinomialData (m: integer, p: real): integer;
function BinomialData (m: integer; p: real; n: integer): intvector;
```

generates binomial data with paramters m and p. The vectorized version generates an integer vector with n independent realizations.

Available in StatPascal

7.12 BoxPlot

```
procedure BoxPlot (x: real; y: realvector; win, des: string)
```

A boxplot of the data contained in y is plotted at the position specified in x in the given window. The description provided in *des* is added to the plot.

Available in StatPascal

```
var i: integer;
begin
 for i := 1 to 20 do
   boxplot (i, GaussianData (500), 'Boxplot', 'Nr ' + str (i))
end.
```

7.13 CBind

```
funcction CBind (...): matrix of type
```

concatenates matrices and vectors (with the same number of rows) columnwise.

See also: RBind (see Section 7.135)

```
program CBindDemo;
var a: matrix of real;
begin
    a := unitmatrix (3);
    writeln (cbind (a, realvect (1, 3, 3), a));
end.
```

7.14 ChiSquareData

```
function ChiSquareData (n: integer): real
function ChiSquareData (n, m: integer): realvector
```

generates data according to a chi-square distribution with n degrees of freedom. The vectorized version generates a real vector with m independent realizations.

Available in UFO, StatPascal

```
const n = 5;
var x: vector of real;
begin
  x := ChiSquareData (n, 100)
end.
```

7.15 ChiSquareDensity

```
function ChiSquareDensity (x: real; n: integer): real
```

returns the density of the chi-square distribution with n degrees of freedom.

Available in UFO, StatPascal

```
const n = 5;
var x: real;
begin
  x := 1.0;
  writeln (ChiSquareDensity (x, n))
end.
```

7.16 ChiSquareDF

```
function ChiSquareDF (x: real; n: integer): real
```

returns the df of the chi-square distribution with n degrees of freedom.

Available in UFO, StatPascal

```
const n = 5;
var x: real;
begin
  x := 1.0;
  writeln (ChiSquareDF (x, n))
end.
```

7.17 ChiSquareQF

`function ChiSquareQF (x: real; n: integer): real`

returns the qf of the chi-square distribution with n degrees of freedom.

Available in UFO, StatPascal

```
const n = 5;
var x: real;
begin
  x := 1.0;
  writeln (ChiSquareQF (x, n))
end.
```

7.18 Chol

`function Chol (A: realmatrix): realmatrix`

returns the Cholesky decomposition of a symmetric, positive definite real matrix A.

Available in StatPascal

7.19 Choose

`function Choose (n, k: integer): intvector`

draws k values from the set $\{1, \ldots, n\}$ without replacement. Note that `Choose (n, n)` will generate a random permutation of the numbers $1, \ldots, n$.

Available in StatPascal

7.20 chr

```
function chr (n: integer): char
```

chr converts an integer representation of a character value to the pertaining character. If *ch* is a character variable, then the following holds: `ch = chr (ord (ch))`.

Available in StatPascal

```
var i: integer;
begin
  for i := ord ('A') to ord ('Z') do
    writeln (i, '  ', chr (i))
end.
```

7.21 ClearWindow

```
procedure ClearWindow (win: string)
```

erases all plots from the given window.

Available in StatPascal

7.22 ColumnData

```
function ColumnData (i: integer): realvector
```

returns a real vector with the *i*-th column of the active data set.
See also: Data (see Section 7.30), RowData (see Section 7.143).

Available in StatPascal

```
var x: vector of real;
    i: integer;
begin
  for i := 1 to dimension do begin
    x := ColumnData (i);
    writeln ('Column i: ', x)
  end
end.
```

7.23 ColumnName

```
function ColumnName (i: integer): string
```

returns the name of the i-th column of the active multivariate data set.

Available in StatPascal

```
var i: integer;
begin
  for i := 1 to dimension do
    writeln ('Name of column i: ', ColumnName (i))
end.
```

7.24 cos

```
function cos (x: real): real;
function cos (x: realvector): realvector
```

returns the cosine of x.

Available in UFO, StatPascal

```
var x: real;
begin
  x := cos (0.7)
end.
```

7.25 cosh

```
function cosh (x: real): real;
function cosh (x: realvector): realvector
```

returns the hyperbolic cosine of x.

Available in UFO, StatPascal

```
var x: real;
begin
  x := cosh (0.7)
end.
```

7.26 CreateMultivariate

```
procedure CreateMultivariate (x: realmatrix; fn, desc, headers: string)
```

Generates a multivariate data set from the data contained in the real matrix x. The data set is written to the specified filename, and a short description is appended. The string *headers* contains the column headers, separated by |. Note that

a two–dimensional array can be given instead of a matrix, because an implicit type conversion is supported.

Available in StatPascal

```
var a: array [1..100, 1..3] of real;
    i, j: integer;
begin
    for i := 1 to 100 do
        for j := 1 to 3 do
            a [i, j] := random;
    createmultivariate (a, 'test.dat', 'Demo sample', '1|2|3')
end.
```

7.27 CreateTimeSeries

```
CreateTimeSeries (t, x: realvector; fn, desc: string)
```

Generates a time series from the data provided in t and x. The data set is written to the specified filename, and a short description is appended.

Available in StatPascal

```
var t, x: realvector;
begin
  t := 1..100;
  x := GaussianData (100);
  CreateTimeSeries (t, x, 'whitenoise.dat', '')
end.
```

7.28 CreateUnivariate

```
procedure CreateUnivariate (x: realvector; fn, desc: string)
```

Generates a univariate data set from the data contained in the real vector x. The data set is written to the specified filename, and a short description is appended.

Available in StatPascal

```
var x: vector of real;
begin
  x := GaussianData (100);
  CreateUnivariate (x, 'gaussian.dat', 'Gaussian Data')
end.
```

7.29 CumSum

```
function CumSum (x: realvector): realvector
```

The *CumSum* function returns the cumulative sum of the real vector x.

Available in StatPascal

```
var x: vector of real;
begin
  x := 1..5;
  writeln (cumsum (x))    (* 1 3 6 10 15 *)
end.
```

7.30 Data

```
function Data (i: integer): real;
function Data (i, j: integer): real
```

The *Data* function is used to access the active data set from an StatPascal program. If a univariate data set is active, then `Data(i)` returns the i-th order statistic of the data set. If a multivariate data set is active, then one must also specify a component to be read. Note that `Data (i, 1)` returns the i-th unordered value of a univariate data set. The program is terminated with a run-time error if an invalid index or component is specified.
See also: ColumnData (see Section 7.22), RowData (see Section 7.143), Dimension (see Section 7.34), SampleSize (see Section 7.147)

Available in StatPascal

```
var i: integer;
begin
  for i := 1 to SampleSize do
    writeln (Data (i))
end.
```

7.31 DataType

```
function DataType: string
```

returns a string describing the type of the active data set, whereby the following values are possible: 'univariate', 'multivariate', 'timeseries', 'censored', 'discrete', 'grouped'; a string 'nothing' is returned if no data set was loaded.

Available in StatPascal

```
begin
  if DataType = 'nothing' then
    writeln ('No data set is active')
  else
    writeln ('Active type: ', DataType)
end.
```

7.32 Date

```
procedure Date (d, m, y: integer)
```

Date retrieves the system date (the year is returned as a 4-digit value).

See also: Time (see Section 7.183)

Available in StatPascal

```
var d, m, y: integer;
begin
  date (d, m, y);
  writeln ('Today is: ', d, '.', m, '.', y)
end.
```

7.33 DialogBox

```
function DialogBox (title, prompt: string; default: realvector): realvector
```

If the program is executed within Xtremes, then *DialogBox* displays a dialog box with the given title that asks for the parameters listed in the prompt. The parameters are separated using `|`. The default values are given in the vector *default*. The function returns a vector with the values entered by the user; it returns an empty vector if the user aborts the dialog.

If the program is executed on the command line, the system prompts for the values using the standard input/output device.

Available in StatPascal

```
var para: realvector;
begin
  para := combine (1.0, 0.0, 2.0);
  para := DialogBox ('GP Parameters', 'gamma|mu|sigma', para);
  if size (para) <> 0 then
    writeln ('You entered: ', para)
  else
```

```
    writeln ('Dialog was cancelled')
end.
```

7.34 Dimension

```
function Dimension: integer
```

returns the dimension of the active data set.

Available in StatPascal

```
begin
  if datatype <> 'nothing' then
    writeln ('Dimension of active data: ', Dimension)
end.
```

7.35 DreesPickandsGP

```
procedure DreesPickandsGP (k: integer; var g, m, s: of real; var r: integer)
procedure DreesPickandsGP (x: realvector; k: integer; var g, m, s: real;
var r: integer)
```

returns Drees-Pickands estimate based on the data set provided in x or on the active data set. The routine requires the following parameters:

x	Real vector containing the data. If this parameter is omitted, then the active data set is used.
k	Number of upper extremes used by the estimator.
g, m, s	Estimated shape, location and scale parameter.
r	Errorcode, see estimator error codes (see Section 7.202).

Available in StatPascal

```
var gamma, mu, sigma: real;
    r: integer;
begin
  DreesPickandsGP (EVData (1, 100), 30, gamma, mu, sigma, r);
  writeln (gamma, ' ', mu, ' ', sigma)
end.
```

7.36 EndMultivariate

```
procedure EndMultivariate (Filename, Description: string)
```

The multivariate data set created using BeginMultivariate (see Section 7.10) and SetColumn (see Section 7.151) is stored under the specified filename and becomes the active data set; the description is added. See SetColumn (see Section 7.151) for an example.

Available in StatPascal

7.37 eof

```
function eof (f: text): boolean;
function eof (f: file): boolean
```

returns *true* if the end of the specified file has been reached.

Availabe in StatPascal

```
var f: text;
    s: string;
begin
  assign (f, 'name.txt');
  reset (f);
  repeat
    read (f, s);
    writeln (s)
  until eof (f);
  close (f)
end.
```

7.38 EstimateBandwidth

```
function EstimateBandwidth (x: vector of real): real;
```

estimates a bandwidth for a kernel density by means of cross validation (using the Epanechnikov kernel).

Available in StatPascal

```
var x: vector of real;
begin
    x := GaussianData (100);
    writeln (EstimateBandwidth (x))
end.
```

7.39 EVData

```
function EVData (gamma: real): real;
function EVData (gamma: real; n: integer): realvector
```

EVData generates standard EV data under the shape parameter *gamma*. The vectorized version generates a real vector with n independent realizations.

Available in UFO, StatPascal

```
const beta = 1.0;
var x: vector of real;
begin
  x := EVData (alpha, 100)
end.
```

7.40 EVDensity

```
function EVDensity (gamma, x: real): real
```

evaluates the standard EV density under the shape parameter *gamma*.

Available in UFO, StatPascal

7.41 EVDF

```
function EVDF (gamma, x: real): real
```

evaluates the standard EV df under the shape parameter *gamma*.

Available in UFO, StatPascal

7.42 EVQF

```
function EVQF (gamma, x: real): real
```

evaluates the standard EV qf under the shape parameter *gamma*.

Available in UFO, StatPascal

7.43 Exists

```
function Exists (b: vector of boolean): boolean
```

returns true iff at least one component of the boolean vector b is true.

Available in StatPascal

```
var x: vector of real;
begin
  x := GaussianData (100);
  if Exists (x > 3) then
    writeln ('At least one value is greater than 3')
end.
```

7.44 exp

```
function exp (x: real): real;
function exp (x: realvector): realvector
```

evaluates the exponential function.

Available in UFO, StatPascal

```
var e: real
begin
  e := exp (1)
end.
```

7.45 ExponentialData

```
function ExponentialData: real;
function ExponentialData (n: integer): realvector
```

generates standard exponential data. The vectorized version generates a real vector with n independent realizations.

Available in UFO, StatPascal

7.46 ExponentialDensity

```
function ExponentialDensity (x: real): real
```

evaluates the standard exponential density.

Available in UFO, StatPascal

7.47 ExponentialDF

```
function ExponentialDF (x: real): real
```

evaluates the standard exponential df.

Available in UFO, StatPascal

7.48 ExponentialQF

```
function ExponentialQF (x: real): real
```

evaluates the standard exponential qf.

Available in UFO, StatPascal

7.49 Extremes

```
function Extremes: integer
```

Extremes returns the number of extremes a user–defined estimator is based on. The function can only be applied in estimator programs written for one of the statistical models `gp, gp1, fullgp` or `fullgp1` within the POT domain.

Available in StatPascal

7.50 Flush

```
procedure flush (f: text);
procedure flush (f: file);
procedure flush
```

flushes buffered output for the given file (or output if no file is specified).

Available in StatPascal

7.51 frac

```
function frac (x: real): real;
function frac (x: realvector): realvector
```

returns $x - [x]$. if x is not negative or $x - [x] + 1$ if x is negative.

Available in UFO, StatPascal

7.52 FrechetData

```
function FrechetData (alpha: real): real;
function FrechetData (alpha: real; n: integer): realvector
```

generates standard Fréchet data under the shape parameter alpha. The vectorized version generates a real vector with n independent realizations.

Available in UFO, StatPascal

7.53 FrechetDensity

```
function FrechetDensity (alpha, x: real): real
```

evaluates the standard Fréchet density under the shape parameter *alpha*.

Available in UFO, StatPascal

7.54 FrechetDF

```
function FrechetDF (alpha, x: real): real
```

evaluates the standard Fréchet df under the shape parameter *alpha*.

Available in UFO, StatPascal

7.55 FrechetQF

```
function FrechetQF (alpha, x: real): real
```

evaluates the standard Fréchet qf under the shape parameter *alpha*.

Available in UFO, StatPascal

7.56 gamma

```
function gamma (x: real): real
```

returns the value of the gamma function $\Gamma(x)$.

Available in UFO, StatPascal

7.57 GammaData

```
function GammaData (r: real): real;
function GammaData (r: real; n: integer): realvector
```

generates data under the gamma distribution with shape parameter r. The vectorized version generates a real vector with n independent realizations.

Available in UFO, StatPascal

7.58 GammaDensity

```
function GammaDensity (r, x: real): real
```

evaluates the density of the gamma distribution with shape parameter r.

Available in UFO, StatPascal

7.59 GammaDF

```
function GammaDF (r, x: real): real
```

evaluates the df of the gamma distribution with shape parameter r. The series expansion is taken from Brandt, S. (1999): *Data Analysis*. Springer, New York.

Available in UFO, StatPascal

7.60 GaussianData

```
function GaussianData: real;
function GaussianData (n: integer): realvector
```

generates standard Gaussian data. The vectorized version generates a real vector with n independent realizations.

Available in UFO, StatPascal

7.61 GaussianDensity

```
function GaussianDensity (x: real): real
```

evaluates the standard Gaussian density.

Available in UFO, StatPascal

7.62 GaussianDF

```
function GaussianDF (x: real): real
```

evaluates the standard Gaussian df.

Available in UFO, StatPascal

7.63 GaussianQF

```
function GaussianQF (x: real): real
```

evaluates the standard Gaussian qf.

Available in UFO, StatPascal

7.64 GCauchyData

```
function GCauchyData (alpha: real): real;
function GCauchyData (alpha: real; n: integer): realvector
```

generates standard generalized Cauchy data under the shape parameter *alpha*. The vectorized version generates a real vector with n independent realizations.

Available in UFO, StatPascal

7.65 GCauchyDensity

```
function GCauchyDensity (alpha, x: real): real
```

evaluates the standard generalized Cauchy density under the shape parameter *alpha*.

Available in UFO, StatPascal

7.66 GCauchyDF

```
function GCauchyDF (alpha, x: real): real
```

evaluates the standard generalized Cauchy df under the shape parameter *alpha*.

Available in UFO, StatPascal

7.67 GCauchyQF

```
function GCauchyQF (alpha, x: real): real
```

evaluates the standard generalized Cauchy qf under the shape parameter *alpha*.

Available in UFO, StatPascal

7.68 GeometricData

```
function GeometricData (p: real): integer;
function GeometricData (p: real; n: integer): intvector;
```

generates geometric data with paramter p. The vectorized version generates an integer vector with n independent realizations.

Available in StatPascal

7.69 GMFEVDF

```
function GMFEVDF (lambda, gamma1, gamma2, x, y: real): real
```

returns the df of the Gumbel-McFadden EV distribution with dependence parameter *lambda* and EV marginals with shape parameter *gamma1* and *gamma2*.

Available in StatPascal

7.70 GMFEVDensity

```
function GMFEVDensity (lambda, gamma1, gamma2, x, y: real): real
```

returns the density of the Gumbel-McFadden EV distribution with dependence parameter *lambda* and EV marginals with shape parameter *gamma1* and *gamma2*.

Available in StatPascal

7.71 GMFEVSF

```
function GMFEVSF (lambda, gamma1, gamma2, x, y: real): real
```

returns the survivor function of the Gumbel-McFadden EV distribution with dependence parameter *lambda* and EV marginals with shape parameter *gamma1* and *gamma2*.

Available in StatPascal

7.72 GMFGPDF

```
function GMFGPDF (lambda, gamma1, gamma2, x, y: real): real
```

returns the df of the Gumbel-McFadden GP distribution with dependence parameter *lambda* and GP marginals with shape parameter *gamma1* and *gamma2*.

Available in StatPascal

7.73 GMFGPDensity

```
function GMFGPDensity (lambda, gamma1, gamma2, x, y: real): real
```

returns the density of the Gumbel-McFadden GP distribution with dependence parameter *lambda* and GP marginals with shape parameter *gamma1* and *gamma2*.

Available in StatPascal

7.74 GMFGPSF

```
function GMFGPSF (lambda, gamma1, gamma2, x, y: real): real
```

returns the survivor function of the Gumbel-McFadden GP distribution with dependence parameter *lambda* and GP marginals with shape parameter *gamma1* and *gamma2*.

Available in StatPascal

7.75 GotoXY

```
procedure GotoXY (f: text; x, y: integer)
```

If the text file f is associated with a text window, then the outpout position is set to the given coordinates. Otherwise, *GotoXY* is ignored.

Available in StatPascal

```
var i: integer;
begin
  for i := 1 to 20 do begin
    GotoXY (output, 2 * i, i);
```

```
    write (i)
  end
end.
```

7.76 GPData

```
function GPData (gamma: real): real;
function GPData (gamma: real; n: integer): realvector
```

returns standard GP data generated under the shape parameter *gamma*. The vectorized version generates a real vector with n independent realizations.

Available in UFO, StatPascal

7.77 GPDensity

```
function GPDensity (gamma, x: real): real
```

returns the standard GP density under the shape parameter *gamma*.

Available in UFO, StatPascal

7.78 GPDF

```
function GPDF (gamma, x: real): real
```

returns the standard GP df under the shape parameter *gamma*.

Available in UFO, StatPascal

7.79 GPQF

```
function GPQF (gamma, x: real): real
```

returns the standard GP qf under the shape parameter *gamma*.

Available in UFO, StatPascal

7.80 GumbelData

```
function GumbelData: real;
function GumbelData (n: integer): realvector
```

generates standard Gumbel data. The vectorized version generates a real vector with n independent realizations.

Available in UFO, StatPascal

7.81 GumbelDensity

```
function GumbelDensity (x: real): real
```

evaluates the standard Gumbel density.

Available in UFO, StatPascal

7.82 GumbelDF

function GumbelDF (x: real): real

evaluates the standard Gumbel df.

Available in UFO, StatPascal

7.83 GumbelQF

```
function GumbelQF (x: real): real
```

evaluates the standard Gumbel qf.

Available in UFO, StatPascal

7.84 HillGP1

```
procedure HillGP1 (k: integer; var a, m, s: real; var r: integer);
procedure HillGP1 (x: realvector; k: integer; var a, m, s: real; var
r: integer)
```

HillGP1 applies the Hill estimator to the active data set. The routine requires the following parameters.

x	Real vector containing the data. If this parameter is omitted, then the active data set is used.
k	Number of upper extremes used by the estimator
a, m, s	Estimated shape, location and scale parameter
r	Errorcode, see estimator error codes (see Section 7.202)

Available in StatPascal

7.85 HREVDF

```
function HREVDF (lambda, gamma1, gamma2, x, y: real): real
```

returns the df of the Huesler-Reiss EV distribution with dependence parameter *lambda* and EV marginals with shape parameter *gamma1* and *gamma2*.

Available in StatPascal

7.86 HREVDensity

```
function HREVDensity (lambda, gamma1, gamma2, x, y: real): real
```

returns the density of the Huesler-Reiss EV distribution with dependence parameter *lambda* and EV marginals with shape parameter *gamma1* and *gamma2*.

Available in StatPascal

7.87 HREVSF

```
function HREVSF (lambda, gamma1, gamma2, x, y: real): real
```

returns the survivor function of the Huesler-Reiss EV distribution with dependence parameter *lambda* and EV marginals with shape parameter *gamma1* and *gamma2*.

Available in StatPascal

7.88 HRGPDF

```
function HRGPDF (lambda, gamma1, gamma2, x, y: real): real
```

returns the df of the Huesler-Reiss GP distribution with dependence parameter *lambda* and GP marginals with shape parameter *gamma1* and *gamma2*.

Available in StatPascal

7.89 HRGPDensity

```
function HRGPDensity (lambda, gamma1, gamma2, x, y: real): real
```

returns the density of the Huesler-Reiss GP distribution with dependence parameter *lambda* and GP marginals with shape parameter *gamma1* and *gamma2*.

Available in StatPascal

7.90 HRGPSF

```
function HRGPSF (lambda, gamma1, gamma2, x, y: real): real
```

returns the survivor function of the Huesler-Reiss GP distribution with dependence parameter *lambda* and GP marginals with shape parameter *gamma1* and *gamma2*.

Available in StatPascal

7.91 Indicator

```
function Indicator (a, b, x: real): integer
```

Indicator returns 1 if x is in the closed interval from a to b, and 0 otherwise.

Available in UFO, StatPascal

```
var x: integer;
begin
  x := Indicator (2, 5, exp (1))
end.
```

7.92 Invert

```
function Invert (A: realmatrix): realmatrix
```

returns the inverse of the real matrix A.

Available in StatPascal

7.93 KernelDensity

```
function KernelDensity (x, b: real; j: integer): real
```

returns the kernel density at the point x for the active data set given the bandwidth

b and the number of the kernel j. The following kernels are available:

$k1(x) = 0.75 * (1.0 - z * *2)$
$k2(x) = 0.5$
$k3(x) = 0.125 * (9.0 - 15.0z * *2)$
$k4(x) = 45/32 * (1 - 10/3z * *2 + 7/3z * *4)$

Available in StatPascal

7.94 log

```
function log (x: real): real;
function log (x: realvector): realvector
```

returns the natural logarithm of the argument.

Available in UFO, StatPascal

7.95 ln

```
function ln (x: real): real;
function ln (x: realvector): realvector
```

returns the natural logarithm of the argument.

Available in StatPascal

7.96 LRSEV

```
procedure LRSEV (var g, m, s: real; var r: integer);
procedure LRSEV (x: realvector; var g, m, s: real; var r: integer)
```

returns the LRSE for the EV model based on the data set provided in x or on the active data set. The routine requires the following parameters:

x	Real vector containing the data. If this parameter is omitted, then the active data set is used.
g, m, s	Estimated shape, location and scale parameter (in gamma parametrization)
r	Errorcode, see estimator error codes (see Section 7.202)

Available in StatPascal

7.97 MakeMatrix

```
function MakeMatrix (a: vector of type; n, m: integer): matrix of type
```

creates a matrix with n rows and m columns and fills it by row with the values given in the vector a. The vector a is repeated cyclically if it is shorter than nm.

Available in StatPascal

```
var a: vector of real;
    B: matrix of real;
begin
  a := combine (2, 5, 1);
  B := MakeMatrix (a, 2, 3);
  writeln (B)    (* 2 5 1
                    2 5 1 *)
end.
```

7.98 max

```
function max (a, b: real): real;
function max (x: realvector): real;
function max (x, y: realvector): realvector
```

max (a, b) returns the maximum of its real arguments. The call *max (x)* returns the maximum component of the vector x, while *max (x, y)* returns a real vector with the componentwise maximum of its arguments x and y.

Available in UFO, StatPascal

7.99 MDEEV

```
procedure MDEEV (var g, m, s: real; var r: integer);
procedure MDEEV (x: realvector; var g, m, s: real; var r: integer)
```

returns the MDE (minimum distance estimator) for the EV model based on the active data set. The routine requires the following parameters.

x	Real vector containing the data. If this parameter is omitted, then the active data set is used.
g, m, s	Estimated shape, location and scale parameter (in gamma parametrization).
r	Errorcode, see estimator error codes (see Section 7.202).

Note: The current implementation minimizes the Hellinger distance between the histogram obtained by grouping the data set into 20 equally sized cells and the EV density.

Available in StatPascal

7.100 MDEGaussian

```
procedure MDEGaussian (var m, s: real; var r: integer;
procedure MDEGaussian (x: realvector; var m, s: real; var r: integer)
```

returns the MDE (minimum distance estimator) for the Gaussian model based on the active data. The routine requires the following parameters.

x	Real vector containing the data. If this parameter is omitted, then the active data set is used.
m, s	Estimated location and scale parameter.
r	Errorcode, see estimator error codes (see Section 7.202).

Available in StatPascal

7.101 Mean

```
function Mean (x: realvector): real
```

returns the mean of the real vector `x`.

Available in StatPascal

7.102 Median

```
function Median (x: realvector): real
```

returns the median of the real vector `x`.

Available in StatPascal

7.103 MEGP1

```
procedure MEGP1 (k: integer; var a, m, s: real; var r: integer;
procedure MEGP1 (x: realvector; k: integer; var a, m, s: real; var r:
integer)
```

MEGP1 applies the M-estimator for the GP1 model to a data set. The routine requires the following parameters.

x	Real vector containing the data. If this parameter is omitted, then the active data set is used.
k	Number of upper extremes used by the estimator.
a, m, s	Estimated shape, location and scale parameter.
r	Errorcode, see estimator error codes (see Section 7.202).

Available in StatPascal

7.104 MemAvail

```
function MemAvail: integer
```

forces a compactification of the heap and returns the number of free data cells. A data cell may store any ordinal value or a real number. It consumes eight bytes of memory on common architectures.

Available in StatPascal

```
begin
  writeln (MemAvail, ' cells available.')
end.
```

7.105 MenuBox

```
function MenuBox (title, prompt: string): integer
```

If the program is executed within Xtremes, then *MenuBox* displays a menu with the given title and the options listed in the prompt. The options are separated using `|`. The function returns an integer with the number of the selected menu option; it returns zero if the user cancels the menu.

If the program is executed on the command line, the system displays the options using the standard input/output device and prompts for a selection.

Available in StatPascal

```
var selection: integer;
begin
  selection := MenuBox ('Select option', 'Option 1|Option 2');
  if selection <> 0 then
    writeln ('Your selection: ', selection)
  else
```

```
    writeln ('Menu was cancelled')
end.
```

7.106 MessageBox

```
procedure MessageBox (s: string)
```

displays the string s in a message box and waits for the selection of the OK-button. The command line version displays the string s and continues execution of the program.

Available in StatPascal

```
var x: real;
begin
  x := sin (0.5);
  MessageBox ('sin (0.5) = ' + str (x))
end.
```

7.107 MHDEGaussian

```
procedure MHDEGaussian (var m, s: real; var r: integer);
procedure MHDEGaussian (x: realvector; var m, s: real; var r: integer)
```

returns the MHDE (minimum Hellinger distance estimator) for the Gaussian model based on the active data. The routine requires the following parameters.

x	Real vector containing the data. If this parameter is omitted, then the active data set is used.
m, s	Estimated location and scale parameter.
r	Errorcode, see estimator error codes (see Section 7.202).

Available in StatPascal

7.108 min

```
function min (a, b: real): real;
function min (x: realvector): real;
function min (x, y: realvector): realvector
```

min (a, b) returns the minimum of its real arguments. The call *min (x)* returns the minimum component of the vector x, while *min (x, y)* returns a real vector with the componentwise minimum of its arguments x and y.

Available in UFO, StatPascal

7.109 MLEEV

```
procedure MLEEV (var g, m, s: real; var r: integer);
procedure MLEEV (x: realvector; var g, m, s: real; var r: integer)
```

returns the MLE for the EV model based on the data set provided in x or on the active data set. The routine requires the following parameters.

x	Real vector containing the data. If this parameter is omitted, then the active data set is used.
g, m, s	Estimated shape, location and scale parameter (in gamma parametrization).
r	Errorcode, see estimator error codes (see Section 7.202).

Available in StatPascal

7.110 MLEEV0

```
procedure MLEEV0 (var m, s: real; var r: integer);
procedure MLEEV0 (x: realvector; var m, s: real; var r: integer)
```

returns the MLE for EV0 (Gumbel) model based on the data set provided in x or on the active data set. The routine requires the following parameters.

x	Real vector containing the data. If this parameter is omitted, then the active data set is used.
m, s	Estimated location and scale parameter.
r	Errorcode, see estimator error codes (see Section 7.202).

Available in StatPascal

7.111 MLEEV1

```
procedure MLEEV1 (var a, m, s: real; var r: integer);
procedure MLEEV0 (x: realvector; var a, m, s: real; var r: integer)
```

returns the MLE for the EV1 (Frechét) model based on the data set provided in x or on the active data set. The routine requires the following parameters.

x	Real vector containing the data. If this parameter is omitted, then the active data set is used.
a, m, s	Estimated shape, location and scale parameter (alpha parametrization).
r	Errorcode, see estimator error codes (see Section 7.202).

Available in StatPascal

7.112 MLEGaussian

```
procedure MLEGaussian (var m, s: real; var r: integer);
procedure MLEGaussian (x: realvector; var m, s: real; var r: integer)
```

returns the MLE for the Gaussian model based on the data set provided in x or on the active data set. The routine requires the following parameters.

x	Real vector containing the data. If this parameter is omitted, then the active data set is used.
m, s	Estimated location and scale parameter.
r	Errorcode, see estimator error codes (see Section 7.202).

Available in StatPascal

7.113 MLEGP

```
procedure MLEGP (k: integer; var g, m, s: of real; var r: integer);
procedure MLEGP (x: realvector; k: integer; var g, m, s: of real; var
r: integer)
```

returns the MLE for the GP model based on the data set provided in x or on the active data set. The routine requires the following parameters.

x	Real vector containing the data. If this parameter is omitted, then the active data set is used.
k	Number of upper extremes used by the estimator.
g, m, s	Estimated shape, location and scale parameter.
r	Errorcode, see estimator error codes (see Section 7.202).

Available in StatPascal

7.114 MLEGP0

```
procedure MLEGP0 (k: integer; var m, s: real; var r: integer);
procedure MLEGP0 (x: realvector; k: integer; var m, s: real; var r: integer)
```

returns the MLE for the GP 0 (exponential) model based on the data set provided in x or the active data set. The routine requires the following parameters.

x	Real vector containing the data. If this parameter is omitted, then the active data set is used.
k	Number of upper extremes used by the estimator.
m, s	Estimated location and scale parameter.
r	Errorcode, see estimator error codes (see Section 7.202)

Available in StatPascal

7.115 Moment

```
function Moment (x: realvector; n: integer): real
```

returns the n-th moment of the real vector `x`.

Available in StatPascal

7.116 MomentGP

```
procedure MomentGP (k: integer; var g, m, s: of real; var r: integer);
procedure MomentGP (x: realvector; k: integer; var g, m, s: of real;
var r: integer)
```

returns the moment estimator for the GP model based on the data set provided in x or on the active data set. The routine requires the following parameters.

x	Real vector containing the data. If this parameter is omitted, then the active data set is used.
k	Number of upper extremes used by the estimator.
g, m, s	Estimated shape, location and scale parameter.
r	Errorcode, see estimator error codes (see Section 7.202).

Available in StatPascal

7.117 NegBinData

```
function NegBinData (m: integer, p: real): integer;
function NegBinData (m: integer; p: real; n: integer): intvector;
```

generates negative binomial data with paramters m and p. The vectorized version generates an integer vector with n independent realizations.

Available in StatPascal

7.118 ord

```
function ord (x): integer;
function ord (x): integervector
```

returns the ordinal value of an ordinal type. If x is a vector with an ordinal base type, then an integer vector is returned.

Available in StatPascal

7.119 Page

```
procedure Page (f: text);
procedure Page
```

Page writes a form feed to the given text (or *output* if no file name is given). If the text file denotes a text window, then the window is cleared.

Available in StatPascal

```
begin
  writeln ('This text will be deleted');
  Page (output)
end.
```

7.120 ParamCount

```
function ParamCount: integer
```

returns the number of arguments provided on the command line. See ParamStr (see Section 7.121) for details.

Available in StatPascal

7.121 ParamStr

```
function ParamStr (i: integer): string
```

returns the i-th argument provided on the command line. Arguments controlling the StatPascal compiler are not accessible.

Assume that the example program below is stored under the filename `printpara.sp`. If you invoke the program with the command

```
sp printpara.sp first second
```

then it will print the words `first` and `second`.

Available in StatPascal

```
var i: integer;
begin
  for i := 1 to ParamCount do
    writeln (ParamStr (i))
end.
```

7.122 ParetoData

```
function ParetoData (alpha: real): real;
function ParetoData (alpha: real; n: integer): realvector
```

generates standard Pareto data under the shape parameter *alpha*. The vectorized version generates a real vector with n independent realizations.

Available in UFO, StatPascal

7.123 ParetoDensity

```
function ParetoDensity (alpha, x: real): real
```

returns the standard Pareto density under the shape parameter *alpha*.

Available in UFO, StatPascal

7.124 ParetoDF

```
function ParetoDF (alpha, x: real): real
```

returns the standard Pareto df under the shape parameter *alpha*.

Available in UFO, StatPascal

7.125 ParetoQF

```
function ParetoQF (alpha, x: real): real
```

returns the standard Pareto qf under the shape parameter *alpha*.

Available in UFO, StatPascal

7.126 Plot

```
procedure Plot (x, y: realvector; win, desc: string)
```

plots linear interpolation of the points (x_1, y_1) to (x_n, y_n) defined by the real vectors x and y within the specified window and adds the specified description.

Available in StatPascal

```
var x, y: realvector;
begin
  x := realvector (0, 2 * 3.14, 100);
  y := sin (x);
  x := cos (x);
  plot (x, y, 'Circle', '')
end.
```

7.127 PlotContour

```
procedure PlotContour (x0, x1, y0, y1: real; v: matrix of real)
```

displays a contour plot of the points $(x_0 + (i-1)/(n-1) * (x_1 - x_0), y_0 + (j-1)/(m-1) * (y_1 - y_0), v[i,j])$ for $1 \leq i \leq n, 1 \leq j \leq m$. n and m are the number of rows and columns of the matrix `v`.

Available in StatPascal

7.128 PlotSurface

```
procedure PlotSurface (x0, x1, y0, y1: real; v: matrix of real)
```

displays a surface plot of the points $(x_0 + (i-1)/(n-1) * (x_1 - x_0), y_0 + (j-1)/(m-1) * (y_1 - y_0), v[i,j])$ for $1 \leq i \leq n, 1 \leq j \leq m$. n and m are the number of rows and columns of the matrix `v`.

Available in StatPascal

7.129 PoissonData

```
function PoissonData (lambda: real): integer;
function PoissonData (lambda: real; n: integer): intvector;
```

generates standard Poisson data under the parameter *lambda*. The vectorized version generates an integer vector with n independent realizations.

Available in StatPascal

7.130 Poly

```
function Poly (a: realvector; x: real): real
```

evaluates the polynomial $a_1 + a_2x + a_3x^2 + ... + a_nx^{n-1}$.

Available in StatPascal

7.131 PolynomialRegression

```
function RegressionPolynomial (x, y: realvector; n: integer): realvector
```

calculates a regression polynomial of degree n based on the data points given in x and y. The coefficients of the polynomial are returned as the result of the function and may be used by the function Poly (see Section 7.130).

Available in StatPascal

```
var x, y, a: vector of real;
begin
  x := GaussianData (100);
  y := 5 * x + 3 + GaussianData (100);
  a := RegressionPolynomial (x, y, 1);
  writeln ('Linear Regression: ', a [2], ' * x + ', a [1])
end.
```

7.132 pred

```
function pred (x)
```

returns predecessor of ordinal type value.

Available in StatPascal

```
begin
  writeln (pred (3))   (* 2 *)
end.
```

7.133 Random

```
function Random: real;
function Random (n: integer): realvector
```

generates [0,1)-uniform data. The vectorized form returns n independent realizations.

Available in UFO, StatPascal

7.134 Rank

```
function Rank (x: realvector): integervector
```

returns the ranks of the values of the real vector x.

Available in StatPascal

7.135 RBind

```
funcction RBind (...): matrix of type
```

concatenates matrices (with the same number of columns) rowwise. A vector is interpreted as n $\times$ 1 matrix and may need to be transposed before using it in *RBind*

See also: CBind (see Section 7.13)

```
program RBindDemo;
var a: matrix of real;
begin
    a := unitmatrix (3);
    writeln (rbind (a, a, transpose (realvector (1, 3, 3))));
end.
```

7.136 read

```
procedure read (var x: T [,...]);
procedure read (f: text; var x: T [,...]);
```

```
procedure read (f: file; var x)
```

read is used to read values from a file. If no file is given as first argument, the text file *input* (which is opened to the standard output device) is used. `T` may be one of the predefined types *integer*, *char*, *real* or *string*. If the first argument is of the type *file*, one must specify a second argument of arbitrary type that is read from the binary file.

Available in StatPascal

```
var x: real;
begin
  read (x);
  writeln (GaussianDensity (x))
end.
```

7.137 ReadData

```
function ReadData (s: string): integer
```

ReadData reads a data set from the file specified by *s*, which becomes the active one. The data set must follow the format of data sets used by Xtremes. The function returns 1 if the data set was read succesfully and 0 if there was an error.

Available in StatPascal

```
begin
  if readdata ('pareto.dat') = 1 then
    write ('Reading of pareto.dat succeeded')
  else
    write ('Reading failed')
end.
```

7.138 readln

```
procedure readln (var x: T [,...]);
procedure readln (f: text; var x: T [,...]);
```

read is used to read values from a file. It performs the same operations like read (see Section 7.136) and advances to the next line of a text file afterwards. Any unread input up to the end of the current line is discarded.

Available in StatPascal.

7.139 RealVect

```
function RealVect (a, b: real; n: integer): realvector
```

returns a real vector with n equally spaced values between a and b.

Available in StatPascal

7.140 Rev

```
function Rev (x: vector): vector
```

reverses the order of elements of a vector.

Available in StatPascal

```
var a: vector of integer;
begin
  a := combine (2, 9, 1, 4);
  writeln (rev (a))         (* 4 1 9 2 *)
end.
```

7.141 RGB

```
function RGB (r, g, b: integer): integer
```

returns a color code to be used in SetColor (see Section 7.150) and SetMarkerFont (see Section 7.158), whereby r, g and b must be between 0 and 255.

7.142 round

```
function round (x: real): integer;
function round (x: realvector): vector of integer
```

rounds a real number to the nearest integer

Available in UFO, StatPascal

```
var n: integer;
begin
  n := round (2.7)   (* 3 *)
end.
```

7.143 RowData

```
function RowData (i: integer): realvector
```

returns a real vector with the i-th row of the active multivariate data set.

See also: Data (see Section 7.30), ColumnData (see Section 7.22).

Available in StatPascal

7.144 SampleDF

```
function SampleDF (x: real): real
```

returns the sample df based on the active univariate data set

Available in UFO, StatPascal

```
var x: real;
begin
  x := SampleDF (2.3)
end.
```

7.145 SampleMeanClusterSize

```
function SampleMeanClusterSize (mingap, u: real): real
```

returns the mean cluster size of the exceedances above the threshold u of the active time series. Two different clusters are separated by a gap consisting of at least `mingap` points.

Available in StatPascal

7.146 SampleQF

```
function SampleQF (x: real): real
```

returns the sample qf based on the active univariate data set.

Available in UFO, StatPascal

```
var x: real;
begin
  x := sampleqf (0.5)
end.
```

7.147 SampleSize

```
function SampleSize: integer
```

returns the size of the active data set.

Available in StatPascal

7.148 SaveEPS

```
procedure SaveEPS (win, filename: string; width, height, left, right,
top, bottom: real)
```

creates an EPS file with the contents of the specified window. The parameters *width* and *height* determine the size of the plot area, while *left*, *right*, *top* and *bottom* specify the size of the border border which is used to print coordinates and additional labels.

Available in StatPascal

```
program SaveEPSDemo;

const
    width = 60;
    height = 44;
    left = 8;
    right = 4;
    top = 4;
    bottom = 8;

    WindowName = 'EPS-Demo';

var
    x, y: vector of real;

begin
    x := realvect (-6, 6, 100);
    y := sin (x);
    plot (x, y, WindowName, 'Sine function');
    SaveEPS (WindowName, 'test.eps', width, height,
             left, right, top, bottom)
end.
```

7.149 ScatterPlot

```
procedure ScatterPlot (x, y: realvector; win: string)
```

displays a scatterplot of the points defined by x and y in a window with the specified title. If the window already exists, then a number is added to the name until a new window can be created.

Available in StatPascal

```
var x, y: realvector;
begin
  x := 1..100;
  y := GaussianData (100);
  ScatterPlot (x, y, 'Gaussian White Noise')
end.
```

7.150 SetColor

```
procedure SetColor (n: integer)
```

sets the color of subsequent plot operations. n may be one of the following values:

ColorBlack ColorRed ColorGreen ColorBlue ColorGrey

Available in StatPascal

7.151 SetColumn

```
procedure SetColumn (col: integer; x: realvector; header: string)
```

The procedure *SetColumn* copies the vector x to the specified column of a multivariate data set created with BeginMultivariate (see Section 7.10). A header is attached to the column.

Available in StatPascal

```
var i: integer;
begin
  BeginMultivariate (100, 3);
  for i := 1 to 3 do
    SetColumn (i, GaussianData (100), 'Column ' + str (i));
  EndMultivariate ('test.dat', 'Description')
end.
```

7.152 SetCoordinates

```
procedure SetCoordinates (win: string; x0, y0, x1, y1: real)
```

sets a new coordinate system in the specified window.

Available in StatPascal.

7.153 SetLabel

```
procedure SetLabel (win: string; mode: integer; x, y: real; s: string)
```

displays a text label `s` in the specified window. The values of `mode`, `x` and `y` define the position of the label and its orientation as well as its aligment. The following constants are defined to build the parameter `mode`; add the desired values (the first entry in each block is equal to zero and can be left out).

VerticalPositionTop	HorizontalPositionRight
VerticalPositionCentered	HorizontalPositionCentered
VerticalPositionBottom	HorizontalPositionLeft
VerticalPositionUser	HorizontalPositionUser
VerticalAlignmentTop	HorizontalAlignmentRight
VerticalAlignmentCentered	HorizontalAlignmentCentered
VerticalAlignmentBottom	HorizontalAlignmentLeft
HorizontalDisplay	
VerticalDisplay	

The values provides for x and y are only used if the flag *HorizontalPositionUser* or *VerticalPositionUser* is specified.

Available in StatPascal

7.154 SetLabelFont

```
procedure SetLabelFont (font, style, size: integer)
```

sets the font for labels created with SetLabel (see Section 7.153).

font is one of the following values: Times Helvetica Courier Symbol

style is one of the following values: Normal Italic Bold BoldItalic

size determines the size of the font used for the labels.

Available in StatPascal

7.155 SetLineOptions

```
procedure SetLineOptions (Thickness, Line, Gap: integer)
```

defines the thickness of a user defined line style as well as the length of line segments and gaps.

Available in StatPascal

7.156 SetLineStyle

```
procedure SetLineStyle (n: integer)
```

sets the line style for subsequent plot operations. *n* may be one of the following values.

SolidLine DashedLine DottedLine DashDottedLine UserLine

If you specify the *UserLine*-style, then the values defined by SetLineOptions (see Section 7.155) are utilized to plot lines.

Available in StatPascal

7.157 SetMarkers

```
procedure SetMarkers (win: string; x0, x1, dx, y0, y1, dy: real)
```

defines the range and the spacing of the markers displayed on the axes of the coordinate system in the specified window.

Available in StatPascal

7.158 SetMarkerFont

```
procedure SetMarkerFont (win: string; font, style, size, color: integer)
```

sets the font for the coordinate system in the given window.

font is one of the following values: Times Helvetica Courier Symbol

style is one of the following values: Normal Italic Bold BoldItalic

size determines the size of the font used for the coordinate system.

color is a value returned by RGB (see Section 7.141) or one of the following values: ColorBlack ColorRed ColorGreen ColorBlue ColorGrey

Available in StatPascal

```
const Window = "CSDemo";
begin
    SetCoordinates (Window, -5, -1, 5, 2);
    SetMarkerFont (Window, Helvetica, BoldItalic, 15, ColorBlack);
end
```

7.159 SetPlotStyle

```
procedure SetPlotStyle (n: integer)
```

selects a plot style for plot operations. *n* may be one of the following values.

LineStyle	Plot solid lines
PointStyle	Plot supporting points using the shape defined with SetPointStyle (see Section 7.160)
BarStyle	Plot vertical bars
BoxStyle	Plot solid line and the supporting points using the shape defined with SetPointStyle (see Section 7.160)
StairCaseStyle	Plot staircases beetween supporting points
AreaStyle	Fill area between graph and the x-axis with the plot color

Available in StatPascal

7.160 SetPointStyle

```
procedure SetPointStyle (n: integer)
```

selects a point style (shape) for plot operations when the plot styles (see Section 7.159) *PointStyle* or BoxStyle are active. The size of the plotted shape is determined by the line thickness (see Section 7.155).*n* may be one of the following values.

Circles	Plot empty circles
FilledCircles	Plot filled circles
Boxes	Plot boxes
Triangles	Plot triangles

Available in StatPascal

7.161 SetTicks

```
procedure SetTicks (win: string; x0, x1, dx, y0, y1, dy: real)
```

defines the range and the spacing of the ticks displayed on the axes of the coordinate system in the specified window.

Available in StatPascal

7.162 sign

```
function sign (x: real): integer;
function sign (x: realvector): integervector
```

returns sign of argument.

Available in UFO, StatPascal

7.163 SimulateRuinTime

```
function SimulateRuinTime (s, rho, beta, lambda, T, gamma, mu, sigma:
real): real
```

The path of a reserve process is simulated and the ruin time (or T if no ruin occurs) is returned. The parameters determine.

s	initial reserve
rho	safety loading
beta	safety exponent
lambda	intensity
T	time horizon
gamma, mu, sigma	GP claim size distribution

Gamma must be less than 1 because otherwise the mean of the GP distribution does not exist.

Available in StatPascal

7.164 sin

```
function sin (x: real): real;
function sin (x: realvector): realvector
```

returns the sine of the argument.

Available in UFO, StatPascal

7.165 sinh

```
function sinh (x: real): real;
function sinh (x: realvector): realvector
```

returns the hyperbolic sine of the argument.

Available in UFO, StatPascal

7.166 size

```
function size (x: vector): integer
```

returns the number of elements of the vector `x`. Strings are internally represented as vectors, so size also determines the length of a string.

Available in StatPascal

```
var x: vector of real;
    s: string;
begin
  x := realvector (0, 1, 11);
  writeln (size (x));            (* 11 *)
  s := 'Hello';
  writeln (size (s))             (* 5 *)
end.
```

7.167 Smooth

```
function Smooth (x: realvector; b: integer): realvector
```

The values of the real vector x are smoothed using a symmetric moving average with equal weights 1/b. The first and last (b+1)/2 values become equal (b should be odd).

Available in StatPascal

7.168 Sort

```
function Sort (x: realvector): realvector
```

The values of the real vector x are sorted.

Available in StatPascal

7.169 sqr

```
function sqr (n: integer): integer;
function sqr (x: real): real;
function sqr (a: integervector): integervector;
function sqr (x: realvector): realvector
```

returns the square of the argument.

Available in UFO, StatPascal

7.170 sqrt

```
function sqrt (x: real): real
function sqrt (x: realvector): realvector
```

returns the square root of the argument.

Available in UFO, StatPascal

7.171 str

```
function str (x: real): string
```

converts a real number to a character string.

Available in StatPascal

```
var s: string;
begin
  s := 'exp (1) = ' + str (exp (1))
end.
```

7.172 StratifiedUniform

```
function StratifiedUniform (n: integer): realvector;
```

generates a stratified sample with n realizations in the interval $[0, 1)$. The returned values will be ordered, with the i-th value being uniform in the interval $[(i - 1)/n, i/n)$, $i = 1, \ldots, n$.

Available in StatPascal

7.173 succ

```
function succ (x: T): T
```

returns the successor of the ordinal type value.

Available in StatPascal

7.174 sum

```
function sum (x: realvector): real
```

returns the sum of the values of the real vector given as argument.

Available in StatPascal

7.175 system

```
function system (s: string): integer
```

passes the string s to the *system* function of the C-library. The result of the call to *system* is returned.

Available in StatPascal

7.176 tan

```
function tan (x: real): real;
function tan (x: realvector): realvector
```

returns the tangent of the argument.

Available in UFO, StatPascal

7.177 tanh

```
function tanh (x: real): real;
function tanh (x: realvector): realvector
```

returns the hyperbolic tangent of the argument.

Available in UFO, StatPascal

7.178 TData

```
function TData (a: real): real
function TData (a: real; n: integer): realvector
```

generates data according to Student's t-distribution with shape parameter a. The vectorized version generates a real vector with m independent realizations.

Available in UFO, StatPascal

7.179 TDensity

```
function TDensity (a, x: real;): real
```

returns the density of Student's t-distribution with shape parameter a.

Available in UFO, StatPascal

```
var x: real;
begin
  x := TDensity (3.0, 1.5)
end.
```

7.180 TDF

```
function TDF (a, x: real;): real
```

returns the df of Student's t-distribution with shape parameter a.

Available in UFO, StatPascal

```
var x: real;
begin
  x := TDF (3.0, 1.5)
end.
```

7.181 TextBackground

```
procedure TextBackground (f: text; color: integer;)
procedure TextBackground (color: integer)
```

sets the background color of subsequent text output.

Available in StatPascal

7.182 TextColor

```
procedure TextColor (f: text; color: integer;)
procedure TextColor (color: integer)
```

sets the color of subsequent text output.

Available in StatPascal

7.183 Time

```
procedure Time (var h, m, s: integer)
```

retrieves system time.

See also: Date (see Section 7.32)

Available in StatPascal

```
var h, m, s: integer;
begin
  time (h, m, s);
   writeln ('The time is ', h, ':', m, '.', s)
end.
```

7.184 TQF

```
function TQF (a, x: real): real
```

returns the qf of Student's t-distribution with shape parameter a.

Available in UFO, StatPascal

```
var x: real;
begin
  x := TQF (3.0, 0.5)
```

```
end.
```

7.185 Transpose

```
function Transpose (A: realmatrix): realmatrix
```

transposes the real matrix A.

Available in StatPascal

7.186 TTest

```
function TTest (x, y: realvector): real
```

returns p-value of two-sided t-test.

Available in StatPascal

7.187 UFODefine

```
procedure UFODefine (s: string)
```

A function or a variable is defined within the UFO-environment of the StatPascal program. These definitions are utilized when the *UFOEvaluate* function is called; they are deleted at the end of the program.

Available in StatPascal

```
begin
  UFODefine ('f(r)=PI*r**2');
  writeln (UFOEvaluate ('f(2)')
end.
```

7.188 UFOMessage

```
function UFOMessage: string
```

returns the error message produced by the last UFO operation. If the operation has been successful, an empty string is returned.

Available in StatPascal

```
var s: string;
    e: real;
```

```
begin
  e := UFOEvaluate ('2*x+');
  s := UFOMessage
end.
```

7.189 UFOEvaluate

```
function UFOEvaluate (s: string): real
```

The string s is evaluated using the UFO facility. One may access functions and variables defined using the integrated calculator or the procedure *UFODefine*.

Available in StatPascal

```
var x: real;
begin
  x := UFOEvaluate ('sin(2)+3')
end.
```

7.190 UniformData

```
function UniformData (m: integer): integer;
function UniformData (m, n: integer): intvector;
```

generates integer uniform data in the set $\{1, \ldots, m\}$. The vectorized version generates an integer vector with n independent realizations.

Available in StatPascal

7.191 UnitMatrix

```
function UnitMatrix (n: integer): realmatrix
```

returns the n-dimensional unit matrix.

Available in StatPascal

```
var A: matrix of real;
begin
  A := UnitMatrix (5);
  writeln (A)
end.
```

7.192 UnitVector

```
function UnitVector (i, n: integer): realvector
```

returns the i-th n-dimensional unit vector.

Available in StatPascal

```
var a: vector of real;
begin
  a := UnitVector (3, 5);
  writeln (a) (* 0 0 1 0 0 *)
end.
```

7.193 Variance

```
function Variance (x: realvector): real
```

returns the empirical variance of the data contained in the real vector x.

Available in StatPascal

```
var x: realvector;
begin
  x := GaussianData (100);
  writeln ('Mean: ', Mean (x), ', variance: ', Variance (x))
end.
```

7.194 WeibullData

```
function WeibullData (alpha: real): real;
function WeibullData (alpha: real; n: integer): realvector
```

generates standard Weibull data under the shape parameter alpha. The vectorized version generates a real vector with n independent realizations.

Available in UFO, StatPascal

7.195 WeibullDensity

```
function WeibullDensity (alpha, x: real): real
```

evaluates the standard Weibull density under the shape parameter *alpha*.

Available in UFO, StatPascal

7.196 WeibullDF

```
function WeibullDf (alpha, x: real): real
```

evaluates the standard Weibull df under the shape parameter *alpha*.

Available in UFO, StatPascal

7.197 WeibullQF

```
function WeibullQf (alpha, x: real): real
```

evaluates the standard Weibull qf under the shape parameter *alpha*.

Available in UFO, StatPascal

7.198 WelchTest

```
function WelchTest (x, y: realvector): real
```

returns the p-value of the two-sided Welch-test.

Available in StatPascal

7.199 WilcoxonTest

```
function WilcoxonTest (x, y: realvector): real
```

returns the p-value of the two-sided Wilcoxon-test.

Available in StatPascal

7.200 write

```
procedure write (x: T [,...])
procedure write (f: text; x: T [,...])
procedure write (f: file; var x)
```

write is used to output values to a file. If no file is given as first argument, the text file *output* (which is opened to the standard output) is used. `T` may be one the predefined data types *boolean*, *char*, *integer*, *real*, a subrange, vector or matrix thereof or a *string*. If the first argument is of the type *file*, one must specify a second argument that is written to the file. Up to three integer expressions can

be appended to an expression that is written to a textfile; they are seperated by semicolons. The first expression denotes the minimum number of characters used to print the value, leading spaces are appended if necessary. If a real value is printed, the second argument defines the number of digits printed after the decimal point. The last expression is used when printing vectors and defines the number of values that are written before a line feed is generated.

Available in StatPascal

```
begin
  writeln ('Phi(2) = ', GaussianDF (2));
  writeln (exp(1):5:2)      (* _2.72 *)
end.
```

7.201 writeln

```
procedure writeln (x: T [,...])
procedure writeln (f: text; x: T [,...])
```

writeln can only be applied to text files. It performs the same operations as *write*, the output position advances to the beginning of the next line after the last character is written.

Available in StatPascal

7.202 Estimator Error codes

The following error codes are returned by an estimator:

0	No error
1	Negative data in GP1 or EV1 model
2	Newton-Raphson iteration found no point of zero
3	k is too small
4	k is larger than sample size
5	Order statistics have different sign (MomentGP only)
6	Sample size is too small
8	Internal error

Appendix A

Porting to StatPascal

A.1 Differences between StatPascal and Pascal

StatPascal supports most features of the Pascal language. The basic data types *boolean*, *char*, *integer*, *real* and *string* (defined as a character vector with special operations) are provided, and new data types can be introduced by means of enumerations, subrange types, arrays, pointers, sets and records. Procedures and functions accept parameters passed by value or by reference. Control structures include assignments, procedure calls, **for**, **while** and **repeat** loops as well as **if** and **case** statements. The following minor changes to Pascal were made.

1. Order of declaration parts: constants, types, variables, functions and procedures may be defined in any order, and more than one declaration of each kind may be given in a block. Note that an object must be defined before it can be used.

   There is one exeption to this rule: To declare a recursive structure, a pointer to a data type declared later is allowed, e.g.,

   ```
   type nodeptr = ^node;
        node    = record
                    contents: integer;
                    next: nodeptr
                  end;
   ```

   The actual type of `nodeptr` is determined at the end of the type declaration part to ensure that `nodeptr` points to the record declared below the pointer. Otherwise, the meaning of a block could be changed by declaring a type `node` in an outer block.

2. Functions may return values of any type. This feature is particularly important as it allows a data set as the result type of a function. The vector structure described in chapter 3 enables the handling of data sets of arbitrary length within subroutines. The result of a function may be defined using the **return** statement which ends the execution immediately. Assignments to the function name are also supported.

   It is also possible to return other functions as result. StatPascal provides functional and procedural data types and considers a subroutine as a constant of such a type.

3. Evaluation of boolean expressions: StatPascal terminates the evaluation of a boolean expression as soon as the result is known. This behaviour can be disabled to achieve compatibility with existing programs relying on side effects.

4. Currently unsupported Pascal constructs: variant records and **with** are not implemented.

5. Typed files are not supported. StatPascal provides two predefined file types for text files (*text*) and binary files (*file*). Values of all types can be written to the latter ones. By default, input and output operations are performed through the predefined text files *input* and *output*.

6. Treatment of pointers and arrays: an optional second argument in a call of *new* allocates multiple values that are addressed by indexing the pointer. We only provide a short example.

```
var p: ^real;
begin
    new (p, 20);    (* allocate p [0], ..., p [19] *)
    dispose (p)     (* release memory *)
end;
```

   The use of these low level features should be avoided. It is preferred to employ the vector structure (see Section 3) of StatPascal when dealing with data sets of unknown size.

A.2 Differences to XPL

XPL, the predecessor of StatPascal, was introduced in the first edition of "Statistical Analysis." That language closely resembled Pascal and lacked the vector and matrix operations provided by StatPascal. We outline the most important changes required for XPL programs to be executed using StatPascal.

- The predefined routines accepting data sets (like *createunivariate*, *plot*, etc.) now require a vector instead of an array. StatPascal can convert arrays to vectors; one only has to delete the size parameter required in XPL. For example, one writes

  ```
  createunivariate (x, filename, description)
  ```

  instead of

  ```
  createunivariate (x, n, filename, description)
  ```

  when creating a univariate data set, whereby `n` is the number of elements in `x`.

- Calls to estimators require scalar variable parameters instead of an array to accept the estimates. For example, one must call the MLE(GP), based on the k largest value of a data set x, using

  ```
  mlegp (x, k, gamma, mu, sigma, errcode)
  ```

  instead of

  ```
  mlegp (x, n, k, p, errcode)
  ```

  where p is a real array with three components.

- The predefined procedures `mark` and `release`, which provided a simple heap management, are no longer available. If you used them to release data sets that were allocated dynamically, you might switch to vectors providing an automatic memory management. The predefined procedure `dispose` is still available.

With these modifications, it should be possible to execute XPL programs under StatPascal. However, it is preferrable to use the new vector and matrix operations of StatPascal because they will result in a higher execution speed of your programs.