OVERVIEW

PASTERP is a PASCAL-like interpreter, with an embedding interface to Borland PASCAL programs. Now you can enhance your applications by providing an easy to use and powerful extension language to your projects.

The language parsed by the interpreter is a subset of PASCAL, with syntax enhancements that make it easier to read. In most of the constructs the PASTERP language is also more forgiving than PASCAL.

This document describes the language syntax, the supported run-time library, and the interface provided to embed the language in your application projects.

Related Topics :

PASTERP Language
PASTERP Run-Time Library
PASTERP <-> Borland PASCAL Interface

PASTERP Language

The PASTERP Language is based on PASCAL with some enhancements that simplify parsing (both for the machine and the users), and is more forgiving than PASCAL.

The language support is described in the following sections :

To understand the language capabilities please refer to the Run-Time Library as well.

STATEMENTS

The following statements are recognized in the PASTERP Language :

GLOBAL Statement

The GLOBAL statement is used to define a global variable, that can be accessed from all the procedures of the PASTERP program being executed. It is important to call the GLOBAL statement only ONCE, or the interpreter will not be able to recognize it as a valid variable definition.

It is a good idea to call this statement in an initialization routine that can be modified by your users.

The GLOBAL statement syntax is as follows :

```
GLOBAL Var-Name : Var-Type [= Initialization-Value] [;]
        [Var-Name ... ]
ENDVAR
```

Where Var-Name is the name of the variable, Var-Type is the type of the variable, and optionally, an Initialization-Value can be specified, using an expression.

Multiple Var-Names can be specified, each one of them will be allowed only if there is no previous variable defined with the same name.

Please note that since PASTERP is an interpreted language, the optional Initialization-Value can be an expression that references functions, variables etc.., however, if you want to translate your PASTERP sources to PASCAL, you should restrict yourself to constant Initialization-Values only.

Related Topics :
LOCAL Statement
Expressions
Variables

LOCAL Statement

The LOCAL statement is used to define a Local variable to the currently executing procedure, that variable can NOT be accessed from any other procedure of the PASTERP program being executed. It is important to call the LOCAL statement only ONCE in the procedure, or the interpreter will not be able to recognize it as a valid variable definition.

It is a good idea to call this statement in an initialization part of your routine, and use it later. The LOCAL statement is closer to the C/C++ variable definition that is performed in the code (and not out of it as in PASCAL), however, unlike C/C++, the variable is not local to a block, but to the entire procedure, from the point of it's declaration.

The LOCAL statement syntax is as follows :

```
LOCAL|VAR Var-Name : Var-Type [= Initialization-Value] [;]
        [Var-Name ... ]
ENDVAR
```

For easier translations from PASCAL to PASTERP, the keyword VAR can be used instead of LOCAL.

Where Var-Name is the name of the variable, Var-Type is the type of the variable, and optionally, an Initialization-Value can be specified, using an expression.

Multiple Var-Names can be specified, each one of them will be allowed only if there is no previous variable defined with the same name.

Please note that since PASTERP is an interpreted language, the optional Initialization-Value can be an expression that references functions, variables etc.., however, if you want to translate your PASTERP sources to PASCAL, you should restrict yourself to constant Initialization-Values only.

Related Topics :

GLOBAL Statement
Expressions
Variables

WRITE Statement

The WRITE statement is used to write a list of expressions. This statement is very close to the PASCAL Write procedure.

The WRITE statement syntax is :

WRITE([File,]Expr-1 [[,] Expr-2 [[,] Expr-3]])[;]

Where Expr-1, Expr-2 .. are expressions that produce an output. In this version of PASTERP these are Numeric and String Expressions.

The optional File parameter is the name of the file that the output will be directed to. In this version of PASTERP, only TEXT files are supported.

Related Topics :
WRITELN Statement
Expressions

WRITELN Statement

The WRITELN statement is used to write a list of expressions. This statement is very close to the PASCAL Writeln procedure. This statement will write a newline character at the end of the arguments list.

The WRITELN statement syntax is :

WRITELN([File,]Expr-1 [[,] Expr-2 [[,] Expr-3]])[;]

or

WRITELN[;]

Where Expr-1, Expr-2 .. are expressions that produce an output. In this version of PASTERP these are Numeric and String expressions.

The optional File parameter is the name of the file that the output will be directed to. In this version of PASTERP, only TEXT files are supported.

Related Topics :
WRITE Statement
Expressions

Assignment Statement

The ASSIGNMENT statement assigns a value to a variable. The variable to be assigned is called the LVALUE of the assignment, and the expression that is being evaluated is called the RVALUE of the assignment.

The ASSIGNMENT statement Syntax is as follows :

Variable := Expression

Where Variable is a variable defined before, as a GLOBAL or LOCAL variable, or was defined by the application program that set the Variable.

The Expression is a Numeric/String/Logical expression that is legal for the LVALUE variable it will be assigned to.

Related Topics :

Expressions
Varaibles
PASTERP <-> Borland PASCAL Interface

IF Statement

The IF statement is used to choose code execution according to a set of rules that is correct (evaluated to TRUE) when the IF statement is executed. This statement is semantically equal to the PASCAL IF statement.

The IF statement syntax is :

```
IF (Conditional-Expression) [THEN]
        ... commands to do if conditional-expression is evaluated to TRUE
[ELSE
        ... commands to do if conditional-expression is evaluated to FALSE]
ENDIF
```

Where Conditional-Expression is a logical expression that can be evaluated to a Boolean value.

Unlike in PASCAL, every IF statement must end with the ENDIF keyword. An optional ELSE keyword defines the end of the statement block that should be evaluated when the Conditional-Expression is evaluated to TRUE, and the start of the statement block that should be executed if the Conditional-Expression is evaluated to FALSE.

Related Topics :
Expressions

WHILE Statement

The WHILE statement is used to create loops that are executed during the time a specific condition is true. The condition is re-evaluated at the beginning of the loop, and if the logic evaluation returns TRUE, a block of commands is executed, until a ENDWHILE (or WEND) keyword is reached.

The WHILE statement syntax is :

```
WHILE (Conditional-Expression)
        ... block of statements
ENDWHILE
```

Where Conditional-Expression is a logical expression that can be evaluated to a Boolean value.

Related Topics :

Expressions
REPEAT Statement
FOR Statement

FOR Statement

The FOR statement is used to loop through a block of instructions a fixed number of times.

The Start/End and Step conditions of the loop are evaluated only once, when the FOR statement starts, this is different from the WHILE and REPEAT statement that are re-evaluated with each iteration.

This statement is close to the standard PASCAL FOR statement. It adds a STEP parameter that defines how the loop's control variable is incremented/decremented. Notice that PASTERP Pascal can use REAL (Floating Point) Variables as control variables.

The FOR statement syntax is :

```
FOR Control-Variable := Start-Value TO|DOWNTO End-Value [STEP Step-Value]
        ... Block of statement
ENDFOR
```

Where Control-Variable is the Numeric variable that will be used as a control variable for the loop, Start-Value is the initial value assigned to the control variable, End-Value is the value that the Control-Variable will be tested against. If the optional Step-Value is supplied, this is the value that will be added to the Control-Variable.

The TO and DOWNTO keywords are used for the same purpose, if the STEP parameter is specified, the Step-Value sets the value that will be added to the Control-variable. If the STEP parameter is not supplied, using TO will assign 1.0 to the Step-Value, and DOWNTO will assign -1.0 to this value.

Related Topics :
Expressions
REPEAT Statement
WHILE Statement

REPEAT Statement

The REPEAT statement is used to create loops that are executed during the time a specific condition is false. The condition is re-evaluated at the end of the loop, and if the logic evaluation returns FALSE, a block of commands is executed.

This statement is different from the WHILE statement, because the command block will be performed at least once, until the first time the conditional-expression is evaluated. In the WHILE statement, the command block might not be executed even once.

The REPEAT statement syntax is :

```
REPEAT
        ... block of statements
UNTIL (Conditional-Expression)
```

Where Conditional-Expression is a logical expression that is re-evaluated at the end of the loop, and the loop is executed while it is evaluated to FALSE.

Related Topics :

Expressions
WHILE Statement
FOR Statement

READ Statement

The READ statement is used to get input from the keyboard, or a file. Unlike the PASCAL READ statement, PASTERP READ statement receives only one argument to read.

The READ statement syntax is :

READ([File,] Variable)[;]

Where Variable is the variable the data will be read into. The optional File parameters is the source file of the input, if no file is specified, the input is received from the keyboard, otherwise, it arrives from the specified file. In this version of PASTERP the only files supported are TEXT files.

Related Topics :

Variables
READLN Statement

READLN Statement

The READLN statement is used to get input from the keyboard, or a file. Unlike the PASCAL READLN statement, PASTERP READLN statement receives only one argument to read.

The READLN statement syntax is :

READLN([File,] Variable)[;]

Where Varaible is the varaible the data will be read into. The optional File parameters is the source file of the input, if no file is specified, the input is received from the keyboard, otherwise, it arrives from the specified file. In this version of PASTERP the only files supported are TEXT files.

Related Topics :

Variables
READ Statement

Procedure Call Statement

PROCEDURE (and FUNCTIONS) CALL are recognized as statements by the PASTERP language. When a procedure call is recognized, the PASTERP interpreter passes control to the specified procedure/function, and continues execution in that function/procedure. When the called procedure exits, execution is resumed after the call to the procedure/function.

The PROCEDURE/FUNCTION CALL syntax is :

Procedure-Name[(Parameter-1, Parameter-2)]

Where Procedure-Name is the name of the procedure/function, that had been defined either by the calling application, or in the PASTERP code.

The optional Parameters are the parameters defined in the procedure definition.

Related Topics :
Procedure/Function Definition

RETURN Statement

The RETURN statement is used to exit a procedure/function, and according to the function/procedure return type, return a value.

The RETURN statement is close to the C/C++ statement, that has no equivalent in PASCAL.

An alternative to the RETURN statement is to set the function value, by assignment, end exit when the ENDPROC keyword is reached, this method is equivalent to the PASCAL return model.

The RETURN statement syntax is :

RETURN [Expression] [;]

Where Expression is the expression that defines the value the function will return, if the function/procedure does not return a value (return type = void), the expression is not necessary.

An alternate syntax is :

```
FUNCTION myfunc(Parameter-List) : Return-Type
 [... some code]
 myfunc := expression
 [... some code]
ENDPROC
```

Related Topics :
Expressions
Procedure/Function definition

Procedure/Function Definition Statement

There are four (4) types of procedures/functions that PASTERP recognizes, of these two are implemented/registered by the Host Application, one is implemented by PASTERP code, and one will be implemented in a future version of PASTERP by dynamic binding.

This section describes procedures and functions definition that are defined in PASTERP source. The other types are defined elsewhere in this document.

Procedures or functions that are defined in PASTERP source must be defined on a new source line. You can not start a procedure/function definition on a line that has any previous statement, or even remarks.

The following syntax is used to define procedures and functions :

```
PROCEDURE Proc-Name[(Parameters-List)] [BEGIN]
 .. procedure code
ENDPROC
```

or

```
FUNCTION Func-Name[(Parameter-List)] : Return-Type [BEGIN]
 .. function code
ENDPROC
```

Where Proc-Name/Func-Name is the name of the procedure. Please note that this name must be unique, or a problem might occur.

The optional Parameter-List is a list of parameters that should be passed to the procedure/function, using the following syntax :

Parameter-Name : Parameter-Type [, Parameter-Name : Parameter-Type [..]]

Where Parameter-Name is the name the parameter will be called in the procedure, and Parameter-Type is the type of the parameter.

Return-Type in a FUNCTION definition is the type of the result returned by the function.

Please note that unlike in PASCAL, procedures and functions that are recognized in expressions/statements even if they are declared and defined after the procedure/function call. This can be done, because the interpreter updates the internal procedure table while it loads the source file to be interpreted.

Another important issue to notice, is that PASTERP procedures/functions CANNOT be nested in other procedures/functions. This is more like the C/C++ functions scope rules.

Related Topics :
RETURN Statement

CONTINUE Statement

The CONTINUE statement is used to start a new iteration of a WHILE, REPEAT or FOR statement. The CONTINUE statement is evaluated as a ENDWHILE, UNTIL or ENDFOR keyword is for the relevant statements.

If no loop is defined, CONTINUE will result in an error code.

The CONTINUE syntax is :

CONTINUE

Related Topics :

WHILE Statement
FOR Statement
REPEAT Statement

VARIABLES

PASTERP variables must be declared before they can be used. Variables can be declared either in the PASTERP source code, or in the host application.

PASTERP variables are either GLOBAL, where every procedure can access them (they have a global scope), or LOCAL to the procedure that executes them.

GLOBAL variables can be defined either from the PASTERP source, or the host application code, LOCAL variables can be defined only in the PASTERP source code, or as parameters to procedures that can be defined by the host application that registers the procedure/function.

This version of PASTERP supports only the built-in variables types. New types can not be created. Arrays and pointers are not supported in this version of PASTERP.

The supported variable types are :

BYTE	- Equal to PASCAL BYTE, a 0-255 integer type.
INTEGER	- Equal to PASCAL INTEGER, a -32K .. + 32K integer type.
WORD	- Equal to PASCAL WORD, a 0 .. 64K integer type.
LONGINT	- Equal to PASCAL LONGINT, a -2 Billion .. + 2 Billion integer type.
REAL	- Equal to PASCAL REAL, a 2.9*10-39 .. 1.7*10.38 float.
STRING	- Equal to PASCAL STRING, a 255 Character dtring.
PCHAR	- Equal to PASCAL PCHAR, an AsciiZ pointer.
BOOLEAN	- Equal to PASCAL BOOLEAN, a TRUE/FALSE logical variable.
TEXT	- EQUAL to PASCAL TEXT, a text mode file.

Please note that while PASTERP does not support most of the other PASCAL types, the keywords for all the standard PASCAL types are reserved by PASTERP for a future release that might support them.

EXPRESSIONS

PASTERP supports expressions that are either Numeric, String or Logical expressions. These expressions are evaluated by the interpreter according to the type of function return, parameter or variable assignment.

In PASTERP all Numeric expressions are evaluated as REAL expressions, and data is converted back and forth if needed between REALs and the Variable/ Parameter used.

All PASTERP String expressions are evaluated as AsciiZ expressions, and data is converted back and forth if needed between AsciiZ and STRING variables/parameters.

The Expressions Definitions are :

Numeric Expressions

In PASTERP all Numeric expressions are evaluated as REAL expressions, and data is converted back and forth if needed between REALs and the Variable/ Parameter used.

The PASTERP Numeric Expressions will be described in a simple structure :

A Numeric Expression supports the standard math operations (+, -, *, /, %) it also supports the POWER operator, parenthesis, and unary minus.

The Primitive elements of a numeric expression are numeric constants, variables of a numeric type, and functions that return a numeric value.

The operators in decreasing evaluation order are :

Primitives
Parenthesis
Unary Minus
POWER
Mul (*), Div (/), Mod (%)
Add (+), Sub (-)

Operators on the same line are left associative.

Related Topics :

Variables
String Expressions
Logical (Boolean) Expressions

String Expressions

All PASTERP String expressions are evaluated as AsciiZ expressions, and data is converted back and forth if needed between AsciiZ and STRING variables/parameters.

String expressions support string concatenation using the + operator.

The Primitive elements of a string expression are string constants, variables of a string type, and functions that return a string value.

Please note that you can concatenate AsciiZ (PCHAR) and STRING type strings.

String Constants are delimited either by single or double quotes, the matching quote is determined by the first quote, this way it is easy to create strings that include the "other" quote character. Like PASCAL, PASTERP Strings can also include the quote character by doubling it.

e.g. - "This string has a single quote right here : ' "

e.g. - 'And this one has a double quote here : " '

Related Topics :

Variables
Numeric Expressions
Logical (Boolean) Expressions

Logical (Boolean) Expressions

PASTERP logical expressions return a Boolean value - TRUE or FALSE.

The supported logical operators are AND, OR, XOR, NOT and parenthesis.

The Primitive Boolean values are TRUE, FALSE, variables of a Boolean type, and functions that return a Boolean type.

The operators in decreasing evaluation order are :

Primitives
Parenthesis
NOT
AND
XOR
OR

Related Topics :

Variables
Numeric Expressions
String Expressions

PASTERP Library

The PASTERP Standard Library is based on the Standard PASCAL library, with some modifications needed to support the extended PASTERP features, and some procedures/functions missing because PASTERP does not support all the PASCAL features.

Extended library will be supplied in a future version, and will support functions that are more related to the PC environment.

The library support is described in the following sections :

PASTERP Standard Library

The PASTERP Standard Library is based on the Standard PASCAL library, with some modifications needed to support the extended PASTERP features, and some procedures/functions missing because PASTERP does not support all the PASCAL features.

The following functions and procedures are defined in the standard library :

Related Topics :

Extended Library

Standard Library Function : PI

function pi : real;

The PI function returns the PI value.

Standard Library Function : EXP

function exp(r : real) : real;

Returns the exponent of (r).

Standard Library Function : SIN

function sin(r : real) : real;

Returns the Sin of (r).

Standard Library Function : RANDOM

function random(l : longint) : longint;

Returns a LONGINT in the range 0 .. l .

Standard Library Function : COS

function cos(r : real) : real;

Returns the Cos of (r).

Standard Library Function : LN

function ln(r : real) : real;

Return the Ln of (r).

Standard Library Function : LOG10

function log10(r : real) : real;

Returns the log (base 10) of (r).

Standard Library Function : LOG2

function log2(r : real) : real;

Returns the log (base 2) of (r).

Standard Library Function : ABS

function abs(r : real) : real;

Returns the absolute value of (r).

Standard Library Function : ARCTAN

function arctan(r : real) : real;

Returns the Arctan of (r).

Standard Library Function : SQR

function sqr(r : real) : real;

Returns the square of (r).

Standard Library Function : SQRT

function sqrt(r : real) : real;

Returns the square root of (r).

Standard Library Function : TAN

function tan(r : real) : real;

Returns the Tan of (r).

Standard Library Function : COTAN

function cotan(r : real) : real;

Returns the COTAN of (r).

Standard Library Function : ARCSIN

function arcsin(r : real) : real;

Returns the Arcsin of (r).

Standard Library Function : ARCCOS

function arccos(r : real) : real;

Returns the Arccos of (r).

Standard Library Function : CHR

function chr(b : byte) : char;

Returns the CHAR representation of (b).

Standard Library Function : ORD

function ord(c : char) : byte;

Returns the ordinal number (representation) of (c).

Standard Library Function : TRUNC

function trunc(r : real) : longint;

Returns (r), truncated.

Standard Library Function : ROUND

function round(r : real) : longint;

Returns (r), rounded.

Standard Library Function : COPY

function copy(s : string, i : byte, l : byte) : string;

Returns the substring of (s), that start and index (i), for (l) bytes.

Standard Library Function : LENGTH

function length(s : string) : byte;

Returns the length of (s).

Standard Library Function : INSERT

function insert(s : string, var d : string, i : index);

Inserts (s) into (d), after position (i).

Standard Library Function : DELETE

procedure delete(var s : string, i : byte, c : byte) : char;

Delete (c) bytes from position (i) of (s).

Standard Library Function : POS

function pos(s : string, d : string) : integer;

Returns the position of (d) in (s), 0 if not found.

Standard Library Function : VAL

function val(s : string, var r : real) : integer;

Returns the value of (s), in (r). If the function returns 0, the conversion was successful, otherwise it points to the index in (s), where the conversion failed.

Standard Library Function : STR

procedure str(r : real, var s : string);

Returns the string representation of (r) in (s).

Standard Library Function : ASSIGN

procedure assign(t : text, s : string);

Associates the text file (t), with the file name specified in (s).

Please note that PASTERP supports automatic assignment of a file name to a text variable during the variables definition.

The following two code fragments are equivalent :

Figure A :

```
var
        t : text;
endvar
        assign(t, "myfile.txt");
```

Figure B :

```
var
        t : text = "myfile.txt";
endvar
```

Standard Library Function : RESET

function reset(t : text) : byte;

Resets (t) for input, and returns an error code. If the function returns 0, the reset operation was successful

Standard Library Function : CLOSE

function close(t : text) : byte;

Closes the text file (t), and returns an error code. If the function returns 0, no error occurred.

Standard Library Function : APPEND

function append(t : text) : byte;

Opens (t) for output, from the end of the file. Returns an error code. If the function returns 0, no error occurred.

Standard Library Function : REWRITE

function rewrite(t : text) : byte;

Opens (t) for output, rewriting over any previous file with the same name. The function returns an error code, or 0 if no error occurred.

Standard Library Function : EOF

function eof(t : text) : Boolean;

Returns TRUE if the file pointer of (t) is at the end of file.

Standard Library Function : INC

procedure inc(var v[, by : real]);

Increments the variable (v) that must be of a numeric type. If the optional (by) parameter is specified, (v) is incremented using (by). Otherwise, (by) is assumed to be 1.

Standard Library Function : DEC

procedure dec(var v[, by : real]);

Decrement the variable (v) (must be of a numeric type). If the optional (by) parameter is specified, (v) is decremented using (by). Otherwise, (by) is assumed to be 1.

PASTERP Extended Library

Extended library will be supplied in a future version, and will support functions that are more related to the PC environment.

The following functions and procedures are defined in the extended library :

Related Topics :
Standard Library

PASTERP Interface to host language

The PASTERP language is designed to be embedded as an extension interpreted language to application. The first target for host embedding are Borland PASCAL applications. This section describes the PASCAL interface between a host application and the PASTERP objects.

The PASTERP interface is described in the following sections :

Overview of PASTERP <-> PASCAL interface

The PASTERP extension language is implemented as a hierarchy of PARSER object classes in Borland Pascal with Objects.

The base object of the hierarchy is called basicParser, because it provides that basic operation of the PASTERP language interpreter.

This object includes the code for scanning the input source files, building the internal data representations, and dispatching the code using a recursive decent parser.

The most basic PASTERP enabled applications will initialize a basicParser object instance, attach a source file to it, and call it to execute macros at different points of the application's execution.

The basicParser object class defines the interface used to register new functions and procedures to the run-time version of the parser.

Future versions of the PASTERP development kit will offer extended parser object classes that will offer extended functionality. The extended library features that will appear in a future version of PASTERP, will be implemented as a descendent parser, that adds these functions. Please note that you can create descendent parsers with your application's specific functions with this version of the PASTERP development kit.

Please note that the information presented in this electronic document is partial, and that the complete parser object class description is presented with the PASTERP Development Kit documentation.

Initializing a PASTERP instance

The basicParser object class defines the following constructors and procedures used for initializing a PASTERP instance.

```
constructor init;
constructor initFile(const s : string);
procedure loadFile(const s : string);
```

The init constructor initializes the basicParser internal data structures and run time tables.

The initFile constructor calls init, and than calls loadFile.

The loadFile procedure reads a source PASTERP file to the internal basicParser source structures, and build the preliminary procedure call structure.

Registering Procedures and Functions

The basicParser object class defines the following methods that are used to register functions that are provided by the host application :

```
procedure registerProc(n : strID; returnType : word;
                       procAddress : procedureVar);
procedure registerProcParm(n : strID; parmType : word;
                           byReference : Boolean);
```

The registerProc method receives a name for the procedure (n), the return type of the procedure/function (returnType), and the address of the procedure that handles the function.

The registerProcParm is used to define the parameters that the host application procedure needs to receive from the calling PASTERP procedure. The procedure name (n), the type of the parameter (parmType) and whether the parameter is passed by value or reference (byRefence) should be specified.

If multiple parameters can be passed to the host application procedure, calls to the registerProcParm must be created in the order of the parameters that should be passed to the procedure.

example :

Let's assume that we are writing yet another text editor, and we want the extension language, implemented using PASTERP, to be able and call a special goto-position procedure, and a function that returns the current line the cursor is on.

We will define the GOTOPOS procedure of our application to PASTERP using :

```
myParser.registerProc('gotopos', ftVoid, fnGotoPos);
myParser.registerProcParm('gotopos', ftLongint, false);
myParser.registerProcParm('gotopos', ftLongint, false);
```

Here we define a function that returns void called gotopos, the function is implemented by an internal procedure in out application, called fnGotoPos

We define two parameters of type longint to the procedure, and both of them are passed by value.

We will now define the CURRENTLINE function of our application :

```
myParser.registerProc('currentline', ftLongint, fnCurrentLine);
```

Here we defined a function called currentline that returns a longint value, and is implemented by a PASCAL procedure called fnCurrentLine.

In our PASCAL code, that implements these functions, we use a pointer to an array of pointers called funcParm to access the parameters passed to use by the PASTERP code, and return the result of the PASCAL function (if any), in a global variable called funcReusltTYPE, where TYPE is the type the function returns.

Example :

```
procedure fnGotoPos;
begin
        myEditor.gotoPos(longPtr(funcParm^[1])^, longPtr(funcParm^[2])^);
        funcSuccess := true;
end; { fnGotoPos }
```

In this procedure we called our application's myEditor object's gotoPos method, with two longint parameters that were passed through funcParm. Notice that a type cast to the appropriate parameter type was needed.

The funcSuccess global variable is used to inform the PASTERP interpreter that the function was executed with no errors.

Example :

```
procedure fnCurrentLine;
begin
        funcResultLong := myEditor.getCurrentLine;
        funcSuccess := true;
end; { fnCurrentLine }
```

In here we return a result to funcResultLong, using a call to our editor's object getCurrentLine method.

After we defined these functions as new functions for the PASTERP interpreter, our users can write the following code in the PASTERP language that will do something :

```
procedure whatever
        if (currentLine < 30)
                gotoPos(30, 1)
        endif
endproc
```

Please note that this topic presented just the basics of adding functions and procedures to the PASTERP run-time, a complete documentation is available with the PASTERP development kit documentation.

Extending the PASTERP syntax/system library

There are three ways to extend the PASTERP syntax: modification of the source code, overriding the basicParser parse method and registering system procedures.

Modification of the PASTERP source code is available to people that purchase the PASTERP source code.

Overriding the parse method of the basicParser is a technique that requires a description of the recursive decent implementation of the basicParser object class, and the way the parser maintains scope parameters. This discussion is beyond the scope of this on-line document, but it appears in the PASTERP development kit documentation.

System Procedures are special procedures that are implemented as methods of a parser object that is a descendent of the basicParser object class.

The methods should registered as system procedures, and assigned a unique procedure id, that will be used during the dispatching of the methods.

The power that a system procedure has that a standard registered procedure lacks, is that the system procedure can use all the basicParser scanning methods, to implement lookahead if needed, and to perform special functions according to the type of the arguments.

In the basicParser object class, system library procedures such as inc, and reset are implemented as system procedures.

The inc procedure was implemented as a system procedure, because the PASTERP syntax allows inc to be either

inc(myVar)

or

inc(myVar, 12)

Inc uses the basicParser peekToken, getToken and getNumExpr methods to determine whether myVar should be incremented by 1 (syntax 1), or by 12 (syntax 2).

The reset function was implemented as a system procedure, because in a future version of PASTERP that will support both TEXT and Binary files (and not just TEXT files as of today), reset(f) will be able to perform different functions according to the type of f, after accessing basicParser's internal vars table.

The complete discussion of extending PASTERP by registering and writing system procedures is included with the PASTERP development kit.