# Table of Contents

About ......................................................................................................................... 1

Chapter 1: Getting started with progress-4gl ........................................................... 2
Remarks ......................................................................................................................... 2
Versions ......................................................................................................................... 2
Examples ....................................................................................................................... 2
  Installation or Setup .................................................................................................. 3
  Hello, World! ............................................................................................................ 10
  FizzBuzz .................................................................................................................... 10
  Setting up the environment ..................................................................................... 11
  Creating the "sports2000" demo database from the command line ......................... 12
  Commenting code .................................................................................................... 13
  Program files ........................................................................................................... 14
  Running sports2000 as a service .............................................................................. 14

Chapter 2: Compiling .............................................................................................. 17
Introduction .................................................................................................................. 17
Syntax ............................................................................................................................. 17
Examples ....................................................................................................................... 17
  Application Compiler ............................................................................................... 17
  COMPILE statement ............................................................................................... 21
  COMPILER system handle ....................................................................................... 22

Chapter 3: Conditional statements ........................................................................ 27
Introduction .................................................................................................................. 27
Examples ....................................................................................................................... 27
  IF ... THEN ... ELSE-statement ........................................................................... 27
  CASE ......................................................................................................................... 28
  IF ... THEN ... ELSE-function .............................................................................. 29

Chapter 4: FIND statement ..................................................................................... 31
Introduction .................................................................................................................. 31
Examples ....................................................................................................................... 31
  FIND basic examples ............................................................................................... 31
About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: progress-4gl

It is an unofficial and free progress-4gl ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official progress-4gl.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com
Chapter 1: Getting started with progress-4gl

Remarks

ABL (Advanced Business Language). Earlier known as Progress 4GL.

Progress ABL is a programming language tied to the Progress OpenEdge environment, its database and surrounding utilities. This makes it a "fourth generation" programming language.

Progress ABL is a strongly typed, late-bound, English-like programming language with growing support for object orientation. The compiled code is run by the "AVM" (ABL Virtual Machine).

The language is developed and maintained by the Progress Corporation (formerly Progress Software).

Versions

<table>
<thead>
<tr>
<th>Version</th>
<th>Retired</th>
<th>Note</th>
<th>Release Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.7</td>
<td>tbd</td>
<td></td>
<td>2017-04-04</td>
</tr>
<tr>
<td>11.6</td>
<td>tbd</td>
<td>Latest: 11.6.3</td>
<td>2015-10-01</td>
</tr>
<tr>
<td>11.5</td>
<td>2017-Dec</td>
<td></td>
<td>2014-12-01</td>
</tr>
<tr>
<td>11.4</td>
<td>2017-Aug</td>
<td></td>
<td>2014-08-01</td>
</tr>
<tr>
<td>11.3</td>
<td>2016-Aug</td>
<td></td>
<td>2013-07-01</td>
</tr>
<tr>
<td>11.2</td>
<td>2016-Feb</td>
<td></td>
<td>2013-02-01</td>
</tr>
<tr>
<td>11.1</td>
<td>2014-Feb</td>
<td></td>
<td>2012-06-01</td>
</tr>
<tr>
<td>11.0</td>
<td>2013-Jun</td>
<td></td>
<td>2011-12-01</td>
</tr>
<tr>
<td>10.2B</td>
<td>tbd</td>
<td>Renamed OpenEdge</td>
<td>2009-12-01</td>
</tr>
<tr>
<td>10.1C</td>
<td>2014-Jul</td>
<td></td>
<td>2008-02-01</td>
</tr>
<tr>
<td>10.0B</td>
<td>2006-Mar</td>
<td></td>
<td>2004-08-01</td>
</tr>
<tr>
<td>8.3E</td>
<td>2010-Feb</td>
<td></td>
<td>2001-12-01</td>
</tr>
</tbody>
</table>

Examples

https://riptutorial.com/
Installation or Setup

Installing Progress

Download your distribution from Progress. If you want a demo license you need to contact them. Make sure you download a 64-bit and not a 32-bit tar file (unless you happen to run a 32-bit machine).

Windows

The download will be a zip archive. Unpack it and simply run setup.exe. The installation will be graphical but otherwise exactly like the one described below.

Linux/Unix/HP-UX etc

Put the tar file on your Progress system. Let's say you have it in your home directory:

```
/home/user/PROGRESSFILENAME.tar
```

Extract it:

```
cd /home/user
tar xvf PROGRESSFILENAME.tar
```

It will create a directory named

```
proinst
```

Change directory to another destination and create a temporary directory there. For example:

```
cd /tmp
mkdir proinst116
cd proinst116
```

Once the installation is complete this directory will contain information about the installation as well as files you can save and used for future automatic repetitions of the same installation.

Now run the installationscript (named "proinst" in the directory "proinst"):

```
/home/user/proinst/proinst
```

This will start the installation:

```
+-------------------------------------------------------------------+
|                              Welcome                              |
+-------------------------------------------------------------------+
|                                                                   |
|         WELCOME TO THE OPENEDGE INSTALLATION UTILITY              |
|                                                                   |
| Ensure that you have your completed "Preinstallation Checklist   |
```
Now you will need to insert license keys, company name etc. It's recommended to download an "addendum file" then you can simply press Ctrl+A and use it.

Adding an addendum-file:

After you've added licenses manually or loaded them via a file you can press Ctrl+V to view products to be installed:
Once you're satisfied, press Ctrl+E to continue the installation or Ctrl+C to quit.

If you move on you will have to OK just one more thing:

Press Y to continue or N to go back.

Depending on what you're installing you might need to set up different products during the installation.

Next step is to decide if you want to enable the "OpenEdge Explorer". Y or N. This can be changed later on.
Now you have to decide directories where you want to install Progress as well as primary working directory (basically where you want to store your code). Change these or move on with the defaults. Historically /usr/dlc has always been the default so you might want to change this to something thats unique for this specific version of Progress - that might help when upgrading. Choose a Complete Install (the default).

Once done: choose Continue with Installation using arrow keys and press enter to continue.

If you're not planning any SQL access you can press enter once and remove the * before SQL, otherwise just Continue with Install.

Most likely you do not need the OpenEdge Adapter for Sonic ESB so press N - otherwise you know what to do.
If you plan on using WebSpeed for producing dynamic HTML press \textit{\texttt{y}}, otherwise \textit{\texttt{n}}.

<table>
<thead>
<tr>
<th>Web Server Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select Web Server Type</td>
</tr>
<tr>
<td>Select Web Server Script directory</td>
</tr>
<tr>
<td>Copy the static HTML to docroot</td>
</tr>
<tr>
<td>Continue with Installation</td>
</tr>
<tr>
<td>Cancel</td>
</tr>
<tr>
<td>Quit Installation</td>
</tr>
<tr>
<td>Help</td>
</tr>
</tbody>
</table>

Setting up WebSpeed: Choose \textit{Select Web Server Type} and set it to \texttt{cgi} (most likely anyway). Web server script directory can be set to your servers \texttt{cgi-bin} directory or something like \texttt{/tmp}. Don't copy the static HTML - it's really outdated. Continue!

<table>
<thead>
<tr>
<th>Language Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chinese (Simplified)</td>
</tr>
<tr>
<td>Czech</td>
</tr>
<tr>
<td>Dutch</td>
</tr>
<tr>
<td>English - American</td>
</tr>
<tr>
<td>English - International</td>
</tr>
<tr>
<td>French</td>
</tr>
<tr>
<td>German</td>
</tr>
<tr>
<td>Italian</td>
</tr>
<tr>
<td>Polish</td>
</tr>
<tr>
<td>Portuguese - Brazilian</td>
</tr>
<tr>
<td>Spanish</td>
</tr>
<tr>
<td>Portuguese</td>
</tr>
<tr>
<td>Swedish</td>
</tr>
<tr>
<td>Spanish - Latin</td>
</tr>
<tr>
<td>Make Default</td>
</tr>
<tr>
<td>Continue with Installation</td>
</tr>
<tr>
<td>Cancel</td>
</tr>
<tr>
<td>Help</td>
</tr>
</tbody>
</table>

Choose \textit{English} unless you really need something else, you can actually select more than one - make one default in that case. Continue!

<table>
<thead>
<tr>
<th>International Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select CharacterSet, Collation, Case</td>
</tr>
<tr>
<td>Select a Date Format</td>
</tr>
<tr>
<td>Select a Number Format</td>
</tr>
<tr>
<td>Continue with Installation</td>
</tr>
<tr>
<td>Cancel</td>
</tr>
<tr>
<td>Quit Installation</td>
</tr>
<tr>
<td>Help</td>
</tr>
</tbody>
</table>

Choose \textit{English} unless you really need something else, you can actually select more than one - make one default in that case. Continue!
For the International Settings you should try and match any previous installations to help yourself in the future. Otherwise you can set it to something that fits your own needs. This can be changed in the future. Use UTF-8 if you want.

Leave the defaults for the Web Services adapter URL unless you have a good reason.

Disable user authentication? Most likely \textbf{N} is what you want.
This is the final (but one) screen before installation begins.

If you choose to do this you might want to make sure there isn't a previous install being overwritten.

Installation in process. Takes a minute or two.

a. Set up and start your Web server
   - If you did not select to "Copy static HTML files to Document Root directory", then manually copy the files or set a link.
   - For NSAPI Messenger, edit the "obj.conf" and "start" files on the Web server.

b. Set up the Broker machine.
   - Set environment variables if necessary.
   - Edit the properties file (ubroker.properties), then start Broker.

c. To validate your configuration through the Messenger Administration Page, enter ?WSMAdmin after the Messenger name in a URL.
   (For example, for CGI, http://hostname/cgi-bin/wspd_cgi.sh?WSMAdmin)
   (For example, for NSAPI, http://hostname/wsnsa.dll?WSMAdmin)

See the "OpenEdge Application Server: Administration" guide for details.
Some information about WebSpeed.

Final screen - End the Installation or View the Release Notes.

You are done!

Silent installation

The installation has stored a file named /usr/dlc/install/response.ini (or your installation directory). This file can be used to repeat the exact same installation again in a "silent" install that can be scripted and run without any interaction.

To run a silent install simply do:

```
```

Hello, World!

Once you’ve started your Progress editor of choice (there are a couple of options) simply write:

```
DISPLAY "Hello, World!".
```

And run by pressing the corresponding key or menu item:

On Windows in AppBuilder: F1 (Compile -> Run)
On Linux/Unix in the 4GL editor: F2 (or ctr1+x) (Compile -> Run)
On Windows in Developer Studio: alt+shift+x, followed by G (Run -> Run As Progress OpenEdge Application)

FizzBuzz

Another example of "Hello World" style programs is FizzBuzz.

```
DEFINE VARIABLE i AS INTEGER NO-UNDO.
DEFINE VARIABLE cOut AS CHARACTER NO-UNDO.
DO i = 1 TO 100:
    /* Dividable by 3: fizz */
```
IF i MODULO 3 = 0 THEN
cOut = "Fizz".
/* Dividable by 5: buzz */
ELSE IF i MODULO 5 = 0 THEN
cOut = "Buzz".
/* Otherwise just the number */
ELSE
cOut = STRING(i).
/* Display the output */
DISPLAY cOut WITH FRAME x1 20 DOWN.
/* Move the display position in the frame down 1 */
DOWN WITH FRAME x1.
END.

Setting up the environment

Linux/Unix

Once you have Progress installed it's very easy to run.

You only need a couple of environment variables. The directory where Progress was installed (default /usr/dlc but can be something else) needs to be in the DLC-variable

DLC=/usr/dlc

And you might also want the "bin" subdirectory of DLC in your PATH:

PATH=$PATH:$DLC/bin

Now you're set!

There's also a script installed called proenv that will do this (and a little bit more) for you. It's default location is /usr/dlc/bin/proenv.

Some utilities:

showcfg

This will list all your installed products.

pro

This will start the "Procedure Editor" where you can edit and run your programs.

pro program.p

Will open program.p for editing if it exists. Otherwise an error will be displayed.

pro -p program.p
This will run "program.p". If there's a compiled file (program.r) present it will be run, otherwise it will be temporarily compiled and after that executed. The compiled file will not be saved.

Creating the "sports2000" demo database from the command line

This shows how to create the demo database used in big parts of Progress documentation: sports2000.

This assumes you have installed the Progress products with at least one type of database license.

Run `proenv` script/bat-file that will give you a prompt with all environment variables set.

Create a directory.

This example is for Windows. Directory handling etc might be different in another OS.

```
proenv> cd \nproenv> mkdir db
proenv> cd db
proenv> mkdir sports2000
proenv> cd sports2000
```

Create a sports2000 database using "prodb"

```
proenv> prodb mySportsDb sports2000
```

Syntax of prodb:

```
prodb name-of-new-database name-and-path-of-source-database
```

This will create a database called "mySportsDb" in the current directory. That database is an exact copy of the sports2000 database that's shipped with the Progress install. Since the source sports2000 database is located in the Progress install directory you don't need to specify path.

If you look at the directory content you will see some files:

```
proenv> dir
2017-01-12  20:24         2 228 224 mySportsDb.b1
2017-01-12  20:24         1 703 936 mySportsDb.d1
2017-01-12  20:24            32 768 mySportsDb.db
2017-01-12  20:24             2 951 mySportsDb.lg
2017-01-12  20:07               368 mySportsDb.st
2017-01-12  20:24           327 680 mySportsDb_10.d1
2017-01-12  20:24            65 536 mySportsDb_10.d2
2017-01-12  20:24         1 310 720 mySportsDb_11.d1
2017-01-12  20:24         1 376 256 mySportsDb_11.d2
2017-01-12  20:24         327 680 mySportsDb_12.d1
2017-01-12  20:24            65 536 mySportsDb_12.d2
2017-01-12  20:24         327 680 mySportsDb_7.d1
2017-01-12  20:24            65 536 mySportsDb_7.d2
2017-01-12  20:24         655 360 mySportsDb_8.d1
2017-01-12  20:24         655 360 mySportsDb_8.d2
2017-01-12  20:24            327 680 mySportsDb_9.d1
```
### File naming

<table>
<thead>
<tr>
<th>File name</th>
<th>Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>.db</td>
<td>The main database file. Contains the database schema</td>
</tr>
<tr>
<td>.lg</td>
<td>The database log file. Contains logging information in text format</td>
</tr>
<tr>
<td>.st</td>
<td>The database structure file. Describe the storage layout in a text format</td>
</tr>
<tr>
<td>.d?</td>
<td>The actual data. Different files store data of different formats. The .st file can tell what format</td>
</tr>
</tbody>
</table>

Now you can access the database directly by simply typing `pro mySportsDb`. This will start a Progress Editor that's connected to the database. This will be a single user connection so nobody else will be able to access the database at the same time.

In the editor you can simply type:

```pro
FOR EACH bill NO-LOCK:
   DISPLAY bill.
END.
```

To access the database. Press `Ctrl+X` to execute. This will display all contents of the "bill" table. If you want to cancel you can press `Ctrl+C`.

### Commenting code

```pro
/*
   In all versions of Progress ABL you can write multi line comments
*/

/* They can also span a single line */

//Starting with version 11.6 you can also write single line comments

//Can you nest single line comments? //Yes you can

string = "HELLO". //A single line comment can be written after some code

string2 = "Goodbye". /* And the same thing goes for multi line comments. A difference is that a multi line comment also can precede some code */ i = 1.

/* Is it possible to mix comments? //Yes, but multi line comments always needs to be terminated! */

/* You can nest multi line comments as well
```
/* but then all nested comments must be terminated */ or the compiler
will generate an error */

Formally the single line comment starts with the double slash // and ends with a newline, carriage return or end-of-file.

Program files

Progress ABL code is normally stored in files with different ending depending on what they contain. The endings are optional but rather a defacto standard:

<table>
<thead>
<tr>
<th>Filename extension</th>
<th>Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>.p</td>
<td>A Progress program. Can contain several internal procedures, functions etc</td>
</tr>
<tr>
<td>.i</td>
<td>Include file to be included in other files</td>
</tr>
<tr>
<td>.w</td>
<td>A file containing a graphical representation of a Window or Dialog, WinForm-based.</td>
</tr>
<tr>
<td>.r</td>
<td>The compiled result of any file containing Progress 4GL. Called r-code.</td>
</tr>
<tr>
<td>.cls</td>
<td>A Progress Object Oriented Class</td>
</tr>
<tr>
<td>.wrx</td>
<td>A container for ActiveX data whenever needed (generated by compiling in &quot;AppBuilder&quot;).</td>
</tr>
</tbody>
</table>

To run a program-file in Progress 4GL the **RUN**-statement is used:

RUN program.p. //Will run program.p without parameters.
RUN program.w (INPUT true). //Will run program.w with input parameter set to true.
RUN program. //Will run program.r if present otherwise internal procedure "program".

To include another file in a Progress-program the **{**}-directive is used:

{program.i} //Includes program.i in the current program

Running sports2000 as a service

Once the sports2000 database has been installed it's time to run it as a standalone server (and not connect to it as a file).

Start proenv (proenv in the startmeny on Windows or /usr/install-directory/bin/proenv on Linux/Unix).
This example is from Windows. Linux is the same but you need to change paths etc to match your install.

```
proenv> cd \db\sports2000
proenv> proserve mySportsDb -H localhost -S 9999
OpenEdge Release 11.6 as of Fri Oct 16 19:01:51 EDT 2015
20:09:54 BROKER     This broker will terminate when session ends. (5405)
20:09:54 BROKER     The startup of this database requires 17Mb of shared memory. Maximum
        segment size is 128Mb.
20:09:54 BROKER     0: Multi-user session begin. (333)
20:09:55 BROKER     0: Begin Physical Redo Phase at 0 . (5326)
20:17:36 BROKER     0: Before Image Log Initialization at block 1  offset 5300. (15321)
20:09:55 BROKER     0: Login by xyz on CON:. (452)
20:09:55 BROKER     0: Started for 9999 using TCP IPV4 address 127.0.0.1, pid 2892. (5644)
proenv>
```

(You might not get exactly this output).

This will start the mySportsDb on localhost and use port 9999 as primary port for database access. If you want to connect to this database from another client on the same network or elsewhere localhost wont work. Use your IP-address or hostname instead:

```
proenv> proserve mySportsDb -H 192.168.1.10 -S 9999.
```

**Connecting and disconnecting**

Once your database is up and running you can connect to it in your Progress editor:

```
CONNECT mySportsDb -H localhost -S 9999.
```

or

```
CONNECT "-db mySportsDb -H localhost -S 9999".
```

If you get an error message you have either gotten some information wrong in the command or the database isn't up and running. You could also have a software firewall or similar interfering.

You can check the database logfile (mySportsDb.lg in this example) for any clues.

**Disconnecting is just as easy:**

```
DISCONNECT mySportDb.
```

or

```
DISCONNECT "mySportsDb".
```

**Shutting down the database (or disconnect users)**

To shut the database down you can run the proshut command from proenv:
proenv> proshut mySportsDb
OpenEdge Release 11.6 as of Fri Oct 16 19:01:51 EDT 2015

<table>
<thead>
<tr>
<th>usr</th>
<th>pid</th>
<th>time of login</th>
<th>user id</th>
<th>Type</th>
<th>tty</th>
<th>Limbo?</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>7044</td>
<td>Wed Feb 01 20:22:57 2017</td>
<td>xyz</td>
<td>REMC</td>
<td>XYZ-PC</td>
<td>no</td>
</tr>
</tbody>
</table>

1. Disconnect a User
2. Unconditional Shutdown
3. Emergency Shutdown (Kill All)
)x Exit

1. Use **1** to disconnect specific users.
2. Use **2** to shut down the database. **Note:** no questions asked, shutdown starts directly!
3. Use **3** only if you can't take down the database any other way. This might corrupt your data.
4. Use **x** to exit the proshut utility.

You can also shutdown the database directly from the command line:

proenv>proshut mySportsDb -by

Or disconnect a user from command line (assuming you know it's user number, usr in the list above):

proenv>proshut mySportsDb -C disconnect 24
OpenEdge Release 11.6 as of Fri Oct 16 19:01:51 EDT 2015
User 24 disconnect initiated. (6796)

Read Getting started with progress-4gl online: https://riptutorial.com/progress-4gl/topic/8124/getting-started-with-progress-4gl
Chapter 2: Compiling

Introduction

Compile Progress code as called "r-code" and is normally saved in a file with the extension .r. There are a couple of different ways of compiling: using the \texttt{COMPILE} statement or on Linux or AppBuilder: the built in Application Compiler. Developer Studio (the Eclipse environment) has compiling built into it's build process.

You must have 4GL Development or OpenEdge Studio installed to compile 4GL programs which update the database.

Syntax

- \texttt{COMPILE program.p SAVE.} //Compile program.p and save it's r-code
- \texttt{COMPILE VALUE(var) SAVE.} //Compile the named saved in the variable "var" and save it's r-code
- \texttt{COMPILE prog.p XREF prog.xref LISTING prog.list.} //Compile prog.p and create xref and listing-files. Don't save the r-code.
- \texttt{COMPILE program.p SAVE NO-ERROR.} //Compile program.p, save r-code and suppress errors to stop the execution.

Examples

Application Compiler

Windows AppBuilder

In the Windows Appbuilder the Application Compiler is found in the Tools Menu.
**Procedure Editor (Linux - pro or Windows pro.exe)**

In the Procedure Editor (both Linux and Windows) the Compiler is found in the Tools menu.

**Application Compiler**
Regardless of OS the functionality of the compiler is the same. You can add directories and/or files and compile them.

Main settings (more below):

- Save new .r File. If not checked the files will simply be compiled but not saved. Useful for error tracking for instance.
- Look in Subdirectories. Otherwise subdirectories will have to be added.
- Remove old .r Files. Overwrite old .r file.
- Onlu Compile if No .r File. Only compiles uncompiled files.
Options:

- Propath - shows you the propath and let's you select directories to compile from it.
- Add - lets you input a directory or file.
- Modify - lets you modify an existing entry.
- Delete - Deletes an entry.
- Start Compile - Starts the compiler. Shortcut: F2

The main menu choices:

- File -> Exit: Exits the compiler
- Compile -> Start Compile: Starts the compiler. Shortcut: F2
- Tools -> Access to other tools
- Option -> Compiler... : Settings, se below.
- Help -> OpenEdge Help (Windows Only). Online help. Shortcut: F1

Settings

- Default File Spec: Filename extensions to compile
- Message Log File: File to save messages, warnings and errors in
- Save into: Where to store .r file. If blank the same directory as the code.
- V6Frame: Old and unused...
- Steam-IO: If you want to print the compiler output. Most likely not.
- Listing File: If you want the compiler to create a listing file. Useful for debugging
- Append: add to the existing listing file. Otherwise overwrite.

https://riptutorial.com/
• Xref File: If you want the compiler to create a XREF. Useful for debugging, checking index usage etc.
• XML Format: If the compiler xref should be an xml. Otherwise "plain" text.
• Append: add to the existing xref file. Otherwise overwrite.
• Debug File: Debugger listing file.
• Encryption Key: If the source file is encrypted using xcode insert the key here.
• Minimize R-code Size: Remove some debugging information to keep the r-code small.
• Generate MD-5: Mostly for WebClient compiling.

Basic usage

1. Start the compiler
2. Add a path (if not already saved from last session)
3. Press F2 to compile.
4. Observe any errors.
5. Exit

COMPILE statement

The compile statement lets you compile programs in Progress ABL:

Basic usage:

```
COMPILE hello-world.p SAVE.
```

With a variable:

```
DEFINE VARIABLE prog AS CHARACTER NO-UNDO.
prog = "hello.p".
COMPILE VALUE(prog) SAVE.
```

There are several options to the COMPILE-statement:

```
SAVE states that the .r-code should be saved for future use.
```

```
COMPILE hello-world.p SAVE.
```

```
SAVE INTO dir OR SAVE INTO VALUE(dir-variable) saves the r-code in the specified directory:
```

```
COMPILE hello-world.p SAVE INTO /usr/sources.
```

```
LISTING file. Creates a listing file containing debug information regarding blocks, includes etc.
```

```
COMPILE program.p SAVE LISTING c:\temp\listing.txt.
```

Listing has a couple of options for appending files, page-size and page-width:
XREF xreffile will save a compiler xref file containing information about string and index usage etc. You can also APPEND this one.

```
COMPILE checkFile.p SAVE XREF c:\directory\xref-file.txt.
```

XREF-XML xreffile-or-dir will do the same thing as XREF but save the file in an xml-format instead. If you use a directory the xref-file will be named programname.xref.xml.

```
COMPILE file.p SAVE XREF c:\temp\.
```

NO-ERROR will supress any errors from stopping your program.

```
COMPILE program SAVE NO-ERROR.
```

DEBUG-LIST file generates a debug file with line numbers.

```
COMPILE checkFile.p SAVE DEBUG-LIST c:\temp\debug.txt.
```

PREPROCESS file will first translate all preprocessors and then create a new .p-file with the code prior to compiling.

```
COMPILE checkFile.p SAVE PREPROCESS c:\temp\PREPROC.txt.
```

XCODE key will compile an encrypted source code with key as key. You cannot use XCODE with the XREF, XREF-XML, STRING-XREF, or LISTING options together.

```
COMPILE program.p SAVE XCODE myKey.
```

You can combine several options:

```
COMPILE prog.p SAVE INTO /usr/r-code XREF /usr/xrefs/xref.txt APPEND LISTING /usr/listings.txt APPEND NO-ERROR.
```

**COMPILER system handle**

The COMPILER system handle let's you look at information regarding a recent compile.

Assuming ok-program.p is a program without any errors or warning:

```
COMPILE ok-program.p SAVE NO-ERROR.

DEFINE VARIABLE iError AS INTEGER     NO-UNDO.

MESSAGE
"Errors: " COMPILER:ERROR SKIP
"Warnings: " COMPILER:WARNING SKIP
```
This will proceed:

![Information dialog box with no errors, warnings, or messages.]

Compiling a program with a warning:

```plaintext
/* program-with-warning.p */
DEFINE VARIABLE c AS CHARACTER   NO-UNDO.
DEFINE VARIABLE i AS INTEGER     NO-UNDO.

c = "hello".
DISPLAY c.
//This RETURN makes the program exit here and the code below unreachable.
RETURN.

IF TRUE THEN DO:
i = 10.
END.
```

Compiling the program:

```plaintext
COMPILE program-with-warning.p SAVE.

DEFINE VARIABLE iError AS INTEGER     NO-UNDO.

MESSAGE
"Errors: " COMPILER:ERROR SKIP
"Warnings: " COMPILER:WARNING SKIP
"Messages: " COMPILER:NUM-MESSAGES
VIEW-AS ALERT-BOX INFORMATION.

DO iError = 1 TO COMPILER:NUM-MESSAGES:
    DISPLAY
        COMPILER:GET-FILE-NAME(iError)    LABEL "Filename" FORMAT "x(20)"
        COMPILER:GET-MESSAGE(iError)      LABEL "Message"  FORMAT "x(50)"
        COMPILER:GET-NUMBER(iError)       LABEL "Msg#"
        COMPILER:GET-ERROR-COLUMN(iError) LABEL "Column"
        COMPILER:GET-ERROR-ROW(iError)    LABEL "Row"
    WITH FRAME fr1 SIDE-LABELS 1 COLUMNS.
END.
```

Result:

https://riptutorial.com/
Compiling a program with an error

```plaintext
DEFINE VARIABLE c AS CHARACTER NO-UNDO.
DEFINE VARIABLE i AS INTEGER NO-UNDO.

c = "hello".
DISPLAY c.
//Casting should be required below...
IF TRUE THEN DO:
  i = c.
END.
```

Compiling the program:

https://riptutorial.com/
//Use no-errors to supress any error messages from interrupting us.
COMPILE c:\temp\program-with-error.p SAVE NO-ERROR.

DEFINE VARIABLE iError AS INTEGER     NO-UNDO.

MESSAGE
"Errors: " COMPILER:ERROR SKIP
"Warnings: " COMPILER:WARNING SKIP
"Messages: " COMPILER:NUM-MESSAGES
VIEW-AS ALERT-BOX INFORMATION.

DO iError = 1 TO COMPILER:NUM-MESSAGES:
  DISPLAY
    COMPILER:GET-FILE-NAME(iError) LABEL "Filename" FORMAT "x(20)"
    COMPILER:GET-MESSAGE(iError) LABEL "Message" FORMAT "x(50)"
    COMPILER:GET-NUMBER(iError) LABEL "Msg#"
    COMPILER:GET-ERROR-COLUMN(iError) LABEL "Column"
    COMPILER:GET-ERROR-ROW(iError) LABEL "Row"
  WITH FRAME fr1 SIDE-LABELS 1 COLUMNS 20 DOWN.
DOWN WITH FRAME fr1.
END.

Result, there's almost always two errors per error. "Could not understand" is followed by the actual

![Alert Box](https://riptutorial.com/)

error:
Read Compiling online: https://riptutorial.com/progress-4gl/topic/9029/compiling
Chapter 3: Conditional statements

Introduction

Progress ABL supports two conditional statements: IF/THEN/ELSE and CASE.

Examples

IF ... THEN ... ELSE-statement

In the IF THEN ELSE statement the result can be either a single statement:

```
DEFINE VARIABLE i AS INTEGER     NO-UNDO.
IF i = 0 THEN
  MESSAGE "Zero".
ELSE
  MESSAGE "Something else".
```

Or a block, for instance by adding a DO-block:

```
DEFINE VARIABLE i AS INTEGER     NO-UNDO.
IF i = 0 THEN DO:
  RUN procedure1.
  RUN procedure2.
END.
ELSE DO:
  RUN procedure3.
  RUN procedure4.
END.
```

Several IF-statements can be nested with the ELSE IF-syntax:

```
DEFINE VARIABLE i AS INTEGER     NO-UNDO.
IF i = 0 THEN DO:
  RUN procedure1.
  RUN procedure2.
END.
ELSE IF i = 1 THEN DO:
  RUN procedure3.
  RUN procedure4.
END.
ELSE DO:
  RUN procedure5.
  RUN procedure6.
END.
```

The ELSE-part is not mandatory:
DEFINE VARIABLE l AS LOGICAL  NO-UNDO.

l = TRUE.

IF l = TRUE THEN DO:
    MESSAGE "The l variable has the value TRUE" VIEW-AS ALERT-BOX.
END.

The IF/ELSE IF can compare several conditionals, with or without internal connections. This leaves you free to mess up your code in several ways:

DEFINE VARIABLE i AS INTEGER  NO-UNDO.
DEFINE VARIABLE l AS LOGICAL  NO-UNDO.

IF i < 30 OR l = TRUE THEN DO:
END.
ELSE IF i > 30 AND l = FALSE OR TODAY = DATE("2017-08-20") THEN DO:
END.
ELSE DO:
    MESSAGE "I dont really know what happened here".
END.

CASE

The CASE-statement is a lot more strict than the IF/ELSE-conditional. It can only compare a single variable and only equality, not largert/smaller than etc.

DEFINE VARIABLE c AS CHARACTER NO-UNDO.

CASE c:
    WHEN "A" THEN DO:
        RUN procedureA.
    END.
    WHEN "B" THEN DO:
        RUN procedureB.
    END.
    OTHERWISE DO:
        RUN procedureX.
    END.
END CASE.

Using an OR each WHEN can compare different values:

DEFINE VARIABLE c AS CHARACTER NO-UNDO.

CASE c:
    WHEN "A" THEN DO:
        RUN procedureA.
    END.
    WHEN "B" OR WHEN "C" THEN DO:
        RUN procedureB-C.
    END.
    OTHERWISE DO:

RUN procedureX.
END.
END CASE.

Just like with the `IF`-statement each branch can either be a single statement or a block. Just like with the `ELSE`-statement, `OTHERWISE` is not mandatory.

```plaintext
DEFINE VARIABLE c AS CHARACTER   NO-UNDO.
CASE c:
  WHEN "A" THEN
    RUN procedureA.
  WHEN "B" OR WHEN "C" THEN
    RUN procedureB-C.
END CASE.
```

Unlike a c-style `switch`-clause there’s no need to escape the `CASE`-statement - only one branch will be executed. If several `WHEN`s match only the first one will trigger. `OTHERWISE` must be last and will only trigger if none of the branches above match.

```plaintext
DEFINE VARIABLE c AS CHARACTER   NO-UNDO.
  c = "A".
CASE c:
  WHEN "A" THEN
    MESSAGE "A" VIEW-AS ALERT-BOX. //Only "A" will be messaged
  WHEN "A" OR WHEN "C" THEN
    MESSAGE "A or C" VIEW-AS ALERT-BOX.
END CASE.
```

**IF ... THEN ... ELSE-function**

`IF ... THEN ... ELSE` can also be used like a function to return a single value. This is a lot like the ternary `-? operator of C.

```plaintext
DEFINE VARIABLE i AS INTEGER     NO-UNDO.
DEFINE VARIABLE c AS CHARACTER   NO-UNDO.
  /* Set c to "low" if i is less than 5 otherwise set it to "high"
  c = IF i < 5 THEN "low" ELSE "high".
```

Using parenthesis can ease readability for code like this.

```plaintext
DEFINE VARIABLE i AS INTEGER     NO-UNDO.
DEFINE VARIABLE c AS CHARACTER   NO-UNDO.
  c = (IF i < 5 THEN "low" ELSE "high").
```

The value of the `IF`-part and the value of the `ELSE`-part must be of the same datatype. It’s not possible to use `ELSE IF` in this case.
DEFINE VARIABLE dat AS DATE NO-UNDO.
DEFINE VARIABLE beforeTheFifth AS LOGICAL NO-UNDO.

dat = TODAY.
beforeTheFifth = (IF DAY(dat) < 5 THEN TRUE ELSE FALSE).

Several comparisons can be done in the IF-statement:

DEFINE VARIABLE between5and10 AS LOGICAL NO-UNDO.
DEFINE VARIABLE i AS INTEGER NO-UNDO INIT 7.

between5and10 = (IF i >= 5 AND i <= 10 THEN TRUE ELSE FALSE).
MESSAGE between5and10 VIEW-AS ALERT-BOX.

Read Conditional statements online: https://riptutorial.com/progress-4gl/topic/8904/conditional-statements
Chapter 4: FIND statement

Introduction

The FIND statement is used to retrieve a single record from a single table. It has some limitations compared to FOR or QUERY, but it's a simple and handy statement for fast access to records.

Examples

FIND basic examples

A simple sports2000 example:

```
FIND FIRST Customer NO-LOCK WHERE CustNum = 1 NO-ERROR.
IF AVAILABLE Customer THEN DO:
   DISPLAY Customer.NAME.
END.
ELSE DO:
   MESSAGE "No record available".
END.
```

FIRST - find the first record that matches the query

NO-LOCK - don't lock the record - meaning we will only read and not change the record.

WHERE - this is the query

NO-ERROR - don't fail if there isn't any record available.

(IF) AVAILABLE Customer - check if we found a record or not

```
findLoop:
REPEAT :
   FIND NEXT Customer NO-LOCK WHERE NAME BEGINS "N" NO-ERROR.
   IF AVAILABLE customer THEN DO:
      DISPLAY Customer.NAME.
   END.
   ELSE DO:
      LEAVE findLoop.
   END.
END.
```

Availability and scope

The latest find is always the one the availability check will work against - a unsuccessful find will make AVAILABLE return false.

https://riptutorial.com/
DEFINE TEMP-TABLE tt NO-UNDO
    FIELD nr AS INTEGER.

CREATE tt.
tt.nr = 1.

CREATE tt.
tt.nr = 2.

CREATE tt.
tt.nr = 3.

DISPLAY AVAILABLE tt. // yes (tt with nr = 3 is still available)

FIND FIRST tt WHERE tt.nr = 1 NO-ERROR.
DISPLAY AVAILABLE tt. //yes

FIND FIRST tt WHERE tt.nr = 2 NO-ERROR.
DISPLAY AVAILABLE tt. //yes

FIND FIRST tt WHERE tt.nr = 3 NO-ERROR.
DISPLAY AVAILABLE tt. //yes

FIND FIRST tt WHERE tt.nr = 4 NO-ERROR.
DISPLAY AVAILABLE tt. //no

A record found in "main" will be available in any procedures.

DEFINE TEMP-TABLE tt NO-UNDO
    FIELD nr AS INTEGER.

PROCEDURE av:
    DISPLAY AVAILABLE tt.
    IF AVAILABLE tt THEN DO:
        DISPLAY tt.nr.
    END.
END PROCEDURE.

CREATE tt.
tt.nr = 1.

RUN av. // yes. tt.nr = 1

CREATE tt.
tt.nr = 2.

RUN av. // yes. tt.nr = 2

FIND FIRST tt WHERE tt.nr = 4 NO-ERROR.

RUN av. // no (and no tt.nr displayed)

Also, a record found in a procedure will still be available after that procedure has exited.

DEFINE TEMP-TABLE tt NO-UNDO
    FIELD nr AS INTEGER.

PROCEDURE av:

FIND FIRST tt WHERE tt.nr = 1.
END PROCEDURE.

CREATE tt.
tt.nr = 1.

CREATE tt.
tt.nr = 2.

DISPLAY AVAILABLE tt WITH FRAME x1. // yes.

IF AVAILABLE tt THEN DO:
  DISPLAY tt.nr WITH FRAME x1. //tt.nr = 2
END.

PAUSE.

RUN av.

DISPLAY AVAILABLE tt WITH FRAME x2. // yes.

IF AVAILABLE tt THEN DO:
  DISPLAY tt.nr WITH FRAME x2. //tt.nr = 1
END.

**FIND and locking**

Whenever you FIND a record you can acquire a lock of it.

**NO-LOCK**: Used for read only operations. If you do a FIND <record> NO-LOCK you cannot in any way modify the record.

FIND FIRST Customer NO-LOCK NO-ERROR.

**EXCLUSIVE-LOCK**: Used for updates and deletes. If you do this you will "own" the record and nobody else can modify it or delete it until you're done. They can read it (with no-lock) as long as you haven't deleted it.

FIND FIRST Customer EXCLUSIVE-LOCK NO-ERROR.

**SHARE-LOCK**: Avoid at all cost. This will cause mad headache.

FIND FIRST Customer EXCLUSIVE-LOCK NO-ERROR. //Do this instead.

**UPGRADING your lock from NO-LOCK to EXCLUSIVE-LOCK**

You can easily move from a NO-LOCK to an EXCLUSIVE-LOCK if the need to modify a record has arisen:

FIND FIRST Customer NO-LOCK NO-ERROR.
// Some code goes here
// Now we shall modify
FIND CURRENT Customer EXCLUSIVE-LOCK NO-ERROR.
You can go from EXCLUSIVE-LOCK to NO-LOCK as well.

**LOCKED RECORDS**

Whenever other users might acquire a lock of a record you better take this possibility into account. Collisions will occur!

It's better to handle this programmatically using the `NO-WAIT` statement. This tells the AVM to just pass the FIND if the record is locked by somebody else and let you handle this problem.

```
FIND FIRST Customer EXCLUSIVE-LOCK NO-ERROR NO-WAIT.

/* Check for availability */
IF AVAILABLE Customer THEN DO:

    /* Check that no lock (from somebody else) is present */
    IF NOT LOCKED Customer THEN DO:
        /* Do your stuff here */
        END.
    ELSE DO:
        MESSAGE "I'm afraid somebody else has locked this record!" VIEW-AS ALERT-BOX ERROR.
    END.

END.
```

Read FIND statement online: https://riptutorial.com/progress-4gl/topic/8941/find-statement
Chapter 5: Functions

Introduction

A user defined function in Progress ABL is a reusable program module.

Remarks

- A function must be declared in the "main" procedure. It cannot be declared inside a procedure or inside another function.
- A function in Progress ABL isn't a "first class citizen" unlike in programming languages like Haskell or Javascript. You cannot pass a function as an input or output parameter. You can however invoke them dynamically using `DYNAMIC-FUNCTION` or the `CALL` object.
- Calling functions in your queries can lead to bad performance since index matching will hurt. Try to assign the value of the function to a variable and use that variable in the `WHERE`-clause instead.

Examples

Simple function

/* This function returns TRUE if input is the letter "b" and false otherwise */
FUNCTION isTheLetterB RETURNS LOGICAL (INPUT pcString AS CHARACTER):
  IF pcString = "B" THEN
    RETURN TRUE.
  ELSE
    RETURN FALSE.
END FUNCTION.

/* Calling the function with "b" as input - TRUE expected */
DISPLAY isTheLetterB("b").

/* Calling the function with "r" as input - FALSE expected */
DISPLAY isTheLetterB("r").

Parts of the syntax is actually not required:

/* RETURNS isn't required, INPUT isn't required on INPUT-parameters */
FUNCTION isTheLetterB LOGICAL (pcString AS CHARACTER):
  IF pcString = "B" THEN
    RETURN TRUE.
  ELSE
    RETURN FALSE.
/* END FUNCTION can be replaced with END */
END.

Forward declaring functions
A function can be forward declared, this is similar to specifications in a C header file. That way the compiler knows that a function will be made available later on.

Without forward declarations the function MUST be declared before it's called in the code. The forward declaration consists of the `FUNCTION` specification (function name, return type and parameter data types and order). If the forward declaration doesn't match the actual function the compiler will produce errors and the code will fail to run.

```plaintext
FUNCTION dividableByThree LOGICAL (piNumber AS INTEGER) FORWARD.
DISPLAY dividableByThree(9).
FUNCTION dividableByThree LOGICAL (piNumber AS INTEGER):
  IF piNumber MODULO 3 = 0 THEN
    RETURN TRUE.
  ELSE
    RETURN FALSE.
  END.
END.
```

### Multiple input parameters

/* This will popup a message-box saying "HELLO WORLD" */

```plaintext
FUNCTION cat RETURNS CHARACTER ( c AS CHARACTER, d AS CHARACTER):
  RETURN c + " " + d.
END.
MESSAGE cat("HELLO", "WORLD") VIEW-AS ALERT-BOX.
```

### Multiple return statements (but a single return value)

A function can have multiple return statements and they can be placed in different parts of the actual function. They all need to return the same data type though.

```plaintext
FUNCTION returning DATE ( dat AS DATE):
  IF dat < TODAY THEN DO:
    DISPLAY "<".
    RETURN dat - 200.
  END.
  ELSE DO:
    DISPLAY ">".
    RETURN TODAY.
  END.
END.
MESSAGE returning(TODAY + RANDOM(-50, 50)) VIEW-AS ALERT-BOX.
```

A function actually don't have to return anything at all. Then it's return value will be ? (unknown). The compiler will not catch this (but your colleagues will so avoid it).
/* This function will only return TRUE or ?, never FALSE, so it might lead to troubles */
FUNCTION inTheFuture LOGICAL ( dat AS DATE):
   IF dat > TODAY THEN DO:
      RETURN TRUE.
   END.
END.
MESSAGE inTheFuture(TODAY + RANDOM(-50, 50)) VIEW-AS ALERT-BOX.

Output and input-output parameters

A function can only return a single value but there's one way around that: the parameters are not limited to input parameters. You can declare \texttt{INPUT}, \texttt{OUTPUT} and \texttt{INPUT-OUTPUT} parameters.

Unlike \texttt{INPUT} parameters you must specify \texttt{OUTPUT} or \texttt{INPUT-OUTPUT} before the parameters.

Some coding conventions might not like this but it can be done.

/* Function will add numbers and return a sum (AddSomNumbers(6) = 6 + 5 + 4 + 3 + 2 + 1 = 21
 */
/* It will also have a 1% per iteration of failing
 */
/* To handle that possibility we will have a status output parameter
 */
FUNCTION AddSomeNumbers INTEGER ( INPUT number AS INTEGER, OUTPUT procstatus AS CHARACTER):
   procStatus = "processing".
   DEFINE VARIABLE i AS INTEGER NO-UNDO.
   DEFINE VARIABLE n AS INTEGER NO-UNDO.
   /* Iterate number times */
   DO i = 1 TO number:
      /* Do something */
      n = n + i.
      /* Fake a 1% chance for an error that breaks the function */
      IF RANDOM(1,100) = 1 THEN
         RETURN 0.
      END.
   procStatus = "done".
   RETURN n.
END.

DEFINE VARIABLE ret AS INTEGER NO-UNDO.
DEFINE VARIABLE stat AS CHARACTER NO-UNDO.

/* Call the function */
ret = AddSomeNumbers(30, OUTPUT stat).

/* If "stat" is done we made it! */
IF stat = "done" THEN DO:
   MESSAGE "We did it! Sum:" ret VIEW-AS ALERT-BOX.
END.
ELSE DO:
   MESSAGE "An error occurred" VIEW-AS ALERT-BOX ERROR.
END.
Here's an example of an **INPUT-OUTPUT** parameter:

```plaintext
/* Function doubles a string and returns the length of the new string */
FUNCTION doubleString RETURN INTEGER (INPUT-OUTPUT str AS CHARACTER).
    str = str + str.
    RETURN LENGTH(str).
END.

DEFINE VARIABLE str AS CHARACTER   NO-UNDO.
DEFINE VARIABLE len AS INTEGER     NO-UNDO.
str = "HELLO".
len = doubleString(INPUT-OUTPUT str).
MESSAGE
    "New string: " str SKIP
    "Length: " len VIEW-AS ALERT-BOX.
```

**Recursion**

*See recursion*

A function can call itself and thereby recurse.

```plaintext
FUNCTION factorial INTEGER (num AS INTEGER).
    IF num = 1 THEN
        RETURN 1.
    ELSE
        RETURN num * factorial(num - 1).
    END FUNCTION.
DISPLAY factorial(5).
```

With standard settings (startup parameter) the Progress session won't be able to handle very large numbers in this example. `factorial(200)` will fill the stack and raise an error.

**Dynamic call of a function**

Using **DYNAMIC-FUNCTION** or the **CALL**-object you can dynamically call functions.

```plaintext
DEFINE VARIABLE posY      AS INTEGER     NO-UNDO.
DEFINE VARIABLE posX      AS INTEGER     NO-UNDO.
DEFINE VARIABLE OKkeys    AS CHARACTER   NO-UNDO INIT "QLDRUS".
DEFINE VARIABLE Step      AS INTEGER     NO-UNDO INIT 1.
DEFINE VARIABLE moved     AS LOGICAL     NO-UNDO.
/* Set original position */
posY = 10.
posX = 10.
```

https://riptutorial.com/
/* Move up (y coordinates - steps ) */
FUNCTION moveU LOGICAL (INPUT steps AS INTEGER):
    IF posY = 0 THEN
        RETURN FALSE.
    posY = posY - steps.
    IF posY < 0 THEN
        posY = 0.
    RETURN TRUE.
END FUNCTION.

/* Move down (y coordinates + steps ) */
FUNCTION moveD LOGICAL (INPUT steps AS INTEGER):
    IF posY = 20 THEN
        RETURN FALSE.
    posY = posY + steps.
    IF posY > 20 THEN
        posY = 20.
END FUNCTION.

/* Move left (x coordinates - steps ) */
FUNCTION moveL LOGICAL (INPUT steps AS INTEGER):
    IF posX = 0 THEN
        RETURN FALSE.
    posX = posX - steps.
    IF posX < 0 THEN
        posX = 0.
    RETURN TRUE.
END FUNCTION.

/* Move down (x coordinates + steps ) */
FUNCTION moveR LOGICAL (INPUT steps AS INTEGER):
    IF posX = 20 THEN
        RETURN FALSE.
    posX = posX + steps.
    IF posX > 20 THEN
        posX = 20.
END FUNCTION.

REPEAT:
    DISPLAY posX posY step WITH FRAME x1 1 DOWN.
    READKEY.
    IF INDEX(OKKeys, CHR(LASTKEY)) <> 0 THEN DO:
IF CHR(LASTKEY) = "q" THEN LEAVE.
IF CAPS(CHR(LASTKEY)) = "s" THEN UPDATE step WITH FRAME x1.
ELSE DO:
    moved = DYNAMIC-FUNCTION("move" + CAPS(CHR(LASTKEY)), INPUT step).
    IF moved = FALSE THEN
        MESSAGE "Out of bounds".
    END.
END.
END.

The CALL object is not as lightweight as the DYNAMIC-FUNCTION. It can be used to call different things: functions, procedures, external program, Windows DLL-functions. It can also invoke methods on objects and access getters/setters.

DEFINE VARIABLE functionHandle AS HANDLE      NO-UNDO.
DEFINE VARIABLE returnvalue    AS CHARACTER   NO-UNDO.

FUNCTION isPalindrome LOGICAL (INPUT txt AS CHARACTER, OUTPUT txtBackwards AS CHARACTER):
    DEFINE VARIABLE i AS INTEGER     NO-UNDO.
    DO i = LENGTH(txt) TO 1 BY -1:
        txtBackwards = txtBackwards + SUBSTRING(txt, i, 1).
    END.
    IF txt = txtBackwards THEN
        RETURN TRUE.
    ELSE
        RETURN FALSE.
    END FUNCTION.

CREATE CALL functionHandle.
functionHandle:CALL-NAME      = "isPalindrome".
/* Sets CALL-TYPE to the default */
functionHandle:CALL-TYPE  = FUNCTION-CALL-TYPE.
functionHandle:NUM-PARAMETERS = 2.
functionHandle:SET-PARAMETER(1, "CHARACTER", "INPUT", "HELLO WORLD").
functionHandle:SET-PARAMETER(2, "CHARACTER", "OUTPUT", returnvalue).
functionHandle:INVOKE.
MESSAGE "Text backwards: " returnvalue "Is it a palindrome? " functionHandle:RETURN-VALUE VIEW-AS ALERT-BOX.
DELETE OBJECT functionHandle.

CREATE CALL functionHandle.
functionHandle:CALL-NAME      = "isPalindrome".
/* Sets CALL-TYPE to the default */
functionHandle:CALL-TYPE  = FUNCTION-CALL-TYPE.
functionHandle:NUM-PARAMETERS = 2.
functionHandle:SET-PARAMETER(1, "CHARACTER", "INPUT", "ANNA").
functionHandle:SET-PARAMETER(2, "CHARACTER", "OUTPUT", returnvalue).
functionHandle:INVOKE.
MESSAGE "Text backwards: " returnvalue "Is it a palindrome? " functionHandle:RETURN-VALUE VIEW-AS ALERT-BOX.
DELETE OBJECT functionHandle.

https://riptutorial.com/
Read Functions online: https://riptutorial.com/progress-4gl/topic/8857/functions
Chapter 6: Iterating

Introduction

There are several ways of iterating (looping) in Progress ABL.

Examples

DO WHILE

A DO WHILE loop will continue to loop unless the WHILE-part is met. This makes it easy to run forever and eat up all time from one CPU core.

DO WHILE expression:

END.

expression is any combination of boolean logic, comparisons, variables, fields etc that evaluates to a true value.

/* This is a well defined DO WHILE loop that will run as long as i is lower than 10*/
DEFINE VARIABLE i AS INTEGER     NO-UNDO.
DO WHILE i < 10:
   i = i + 1.
END.
DISPLAY i. // Will display 10

You can use any number of checks in the WHILE-part:

DEFINE VARIABLE i AS INTEGER     NO-UNDO.
DO WHILE TODAY = DATE("2017-02-06") AND RANDOM(1,100) < 99:
   i = i + 1.
END.
MESSAGE i "iterations done" VIEW-AS ALERT-BOX.

However, the compiler wont help you so check that the WHILE-part eventually is met:

/* Oops. Didnt increase i. This will run forever... */
DEFINE VARIABLE i AS INTEGER     NO-UNDO.
DO WHILE i < 10:
   i = 1.
END.

DO var = start TO finish [BY step]

This iteration changes a value from a starting point to an end, optionally by a specified value for
each step. The default change is 1.

```plaintext
DEFINE VARIABLE i AS INTEGER NO-UNDO.
DO i = 10 TO 15:
   DISPLAY i WITH FRAME x1 6 DOWN.
   DOWN WITH FRAME x1.
END.
```

Result:

```
---------- i
  10
  11
  12
  13
  14
  15
```

You can iterate over dates as well:

```plaintext
DEFINE VARIABLE dat AS INTEGER NO-UNDO.
DO dat = TODAY TO TODAY + 3:
END.
```

And over decimals. But then you most likely want to use `BY -` otherwise an `INTEGER` would have done just as fine...

```plaintext
DEFINE VARIABLE de AS DECIMAL NO-UNDO.
DO de = 1.8 TO 2.6 BY 0.2:
   DISPLAY "Value" de.
END.
```

Using `BY a negative number` you can also go from a higher to a lower value:

```plaintext
DEFINE VARIABLE i AS INTEGER NO-UNDO.
DO i = 5 TO 1 BY -1:
END.
```

The expression will be tested until it's no longer met. This makes the counter be higher (if moving upwards) or lower (if moving downwards) once the loop is finished:

```plaintext
DEFINE VARIABLE i AS INTEGER NO-UNDO.
DO i = 5 TO 1 BY -1:
END.
```
MESSAGE i. // Will message 0

Another example:

DEFINE VARIABLE da AS DATE     NO-UNDO.

DISPLAY TODAY. //17/02/06
DO da = TODAY TO TODAY + 10:
END.
DISPLAY da. //17/02/17 (TODAY + 11)

REPEAT

REPEAT, will repeat itself forever unless you explicitly exit the loop:

//Runs forever
REPEAT:
  // Do stuff
END.

To exit the loop you can use the LEAVE keyword. With or without a label. Without a label LEAVE will always effect the current loop. With a name you can specify what loop to LEAVE.

/* Without a label */
REPEAT:
  //Do stuff
  IF TRUE THEN LEAVE.
END.

/* With a label */
loopLabel:
REPEAT:
  //Do stuff
  IF <somecondition> THEN LEAVE loopLabel.
END.

/* Two nested REPEATS */
DEFINE VARIABLE i AS INTEGER     NO-UNDO.
loopLabelOne:
REPEAT:
  loopLabelTwo:
    REPEAT:
      i = i + 1.
      IF RANDOM(1,100) = 1 THEN LEAVE loopLabelTwo.
      IF RANDOM(1,100) = 1 THEN LEAVE loopLabelOne.
    END.
  IF RANDOM(1,100) = 1 THEN LEAVE loopLabelOne.
END.
DISPLAY i.

Read Iterating online: https://riptutorial.com/progress-4gl/topic/9009/iterating
Chapter 7: OS-utilities

Introduction

There are several built-in functions and statements for accessing the operating system.

Examples

**OS-COMMAND**

Executes a OS-command.

OS-COMMAND without any options will start a new shell and not exit it - thus you will on graphical OS:es leave a window "hanging".

```
DEFINE VARIABLE cmd AS CHARACTER   NO-UNDO.

cmd = "dir".

OS-COMMAND VALUE(cmd).
```

There are three options: **SILENT**, **NO-WAIT** and **NO-CONSOLE**.

**SILENT**

After processing an operating system command, the AVM shell pauses. To exit the window in Windows GUI platforms, you must type exit. To exit the window in Windows character platforms, you must type exit and press RETURN or SPACEBAR. You can use the SILENT option to eliminate this pause. Use this option only if you are sure that the program, command, or batch file does not generate any output to the screen. Cannot be used with NO-WAIT.

```
OS-COMMAND SILENT VALUE("runprogram.exe").
```

**NO-WAIT**

In a multi-tasking environment, causes the AVM to immediately pass control back to next statement after the OS-COMMAND without waiting for the operating system command to terminate. Cannot be used with SILENT. This option is supported in Windows only.

```
OS-COMMAND NO-WAIT VALUE("DIR > dirfile.txt").
```

On Linux/Unix you will have to achieve this by preceding the command with a &-sign instead:

```
OS-COMMAND VALUE("ls >> file.txt &").
```
NO-CONSOLE

While processing an operating system command, the AVM creates a console window. The console window may not be cleaned up after the command is executed. You can use the NO-CONSOLE option to prevent this window from being created in the first place.

```
OS-COMMAND NO-CONSOLE VALUE("startbach.bat").
```

No errors are ever returned from `OS-COMMAND` to Progress ABL so you have to check for errors another way, possibly writing them to a file in a shell-script or similar.

OPSYS

The `OPSYS`-function returns what OS the program is running on:

```
MESSAGE OPSYS VIEW-AS ALERT-BOX.
```

Result:

![Message](https://riptutorial.com/46)

It can be used to select what OS-utility to call:

```
IF OPSYS = "LINUX" THEN
   OS-COMMAND VALUE("ls -l").
ELSE
   OS-COMMAND VALUE("dir").
```

OS-ERROR

Returns an error from a previous `OS-*` call represented by an integer. The calls that can return an `OS-ERROR` are:

- OS-APPEND
- OS-COPY
- OS-CREATE-DIR
- OS-DELETE
- OS-RENAME
- SAVE CACHE

Note that `OS-COMMAND` is missing. You need to handle errors in `OS-COMMAND` yourself.
<table>
<thead>
<tr>
<th>Error number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No error</td>
</tr>
<tr>
<td>1</td>
<td>Not owner</td>
</tr>
<tr>
<td>2</td>
<td>No such file or directory</td>
</tr>
<tr>
<td>3</td>
<td>Interrupted system call</td>
</tr>
<tr>
<td>4</td>
<td>I/O error</td>
</tr>
<tr>
<td>5</td>
<td>Bad file number</td>
</tr>
<tr>
<td>6</td>
<td>No more processes</td>
</tr>
<tr>
<td>7</td>
<td>Not enough core memory</td>
</tr>
<tr>
<td>8</td>
<td>Permission denied</td>
</tr>
<tr>
<td>9</td>
<td>Bad address</td>
</tr>
<tr>
<td>10</td>
<td>File exists</td>
</tr>
<tr>
<td>11</td>
<td>No such device</td>
</tr>
<tr>
<td>12</td>
<td>Not a directory</td>
</tr>
<tr>
<td>13</td>
<td>Is a directory</td>
</tr>
<tr>
<td>14</td>
<td>File table overflow</td>
</tr>
<tr>
<td>15</td>
<td>Too many open files</td>
</tr>
<tr>
<td>16</td>
<td>File too large</td>
</tr>
<tr>
<td>17</td>
<td>No space left on device</td>
</tr>
<tr>
<td>18</td>
<td>Directory not empty</td>
</tr>
<tr>
<td>999</td>
<td>Unmapped error (ABL default)</td>
</tr>
</tbody>
</table>

**OS-GETENV function**

Returns the value of any OS environment variable.

```message
MESSAGE OS-GETENV ("OS") VIEW-AS ALERT-BOX.
```

On a Windows machine:
MESSAGE OS-GETENV ("SHELL") VIEW-AS ALERT-BOX.

Result on a Linux machine with Bash as current shell:

```
Message
/bin/bash
<OK>
```

OS-COPY

Copy a file

COPY source-file target-file

Copy `c:\temp\source-file.txt` to `c:\temp\target-file.txt`. You need to check OS-ERROR for success or lack thereof.

```
OS-COPY VALUE("c:\temp\source-file.txt")  VALUE("c:\temp\target-file.txt").
IF OS-ERROR <> 0 THEN DO:
  MESSAGE "An error occurred" VIEW-AS ALERT-BOX ERROR.
END.
```

OS-DELETE

Deletes a file, or a file-tree.

As with many other OS-* utilities, you have to check status in OS-ERROR.

```
OS-DELETE file-or-dir-to-delete [ RECURSIVE ]

Delete the entire /tmp/dir tree:

OS-DELETE VALUE("/tmp/dir") RECURSIVE.

Delete the file called c:\dir\file.txt

OS-DELETE VALUE("c:\dir\file.txt").
```

OS-CREATE-DIR
Creates a directory, status is in \texttt{OS-ERROR}.

\texttt{OS-CREATE-DIR directory}

Create a directory called \texttt{/usr/local/appData}

\texttt{OS-CREATE-DIR VALUE("/usr/local/appData").}

\textbf{OS-APPEND}

Append one file to another. Status is checked in \texttt{OS-ERROR}.

\texttt{OS-APPEND source target}

Appends \texttt{targetfile.txt} with \texttt{sourcefile.txt}:

\texttt{OS-APPEND VALUE("sourcefile.txt") VALUE("targetfile.txt").}

\textbf{OS-RENAME}

Rename a file or directory. Status is in \texttt{OS-ERROR}. Can also be used to move files (or move and rename).

\texttt{OS-RENAME oldname newname}

Rename \texttt{/tmp/old-name} to \texttt{/tmp/new-name}:

\texttt{OS-RENAME VALUE("/tmp/old-name") VALUE("/tmp/new-name").}

Move file \texttt{c:\temp\old.txt} to \texttt{c:\new-dir\old.txt}:

\texttt{OS-RENAME VALUE("c:\temp\old.txt") VALUE("c:\new-dir\old.txt").}

\textbf{OS-DRIVES (Windows only)}

Returns a list of all drives on a system.

\texttt{MESSAGE OS-DRIVES VIEW-AS ALERT-BOX.}

Result with four drives, C through F:

\begin{itemize}
  \item \texttt{Message}
      \begin{itemize}
        \item \texttt{C,D,E,F:}
      \end{itemize}
\end{itemize}
On Linux the list will simply be empty as there by definitions are no "drives" connected. Listing directories is done in another way. Read OS-utilities online: https://riptutorial.com/progress-4gl/topic/9056/os-utilities
Chapter 8: Procedures

Introduction

There are two types of procedures in Progress ABL: internal procedures and procedure prototypes that are facades to Windows dlls or Unix/Linux shared library procedures.

Just like with functions, procedures cannot be nested. You cannot nest functions in procedures and vice versa.

A procedure is called with the run statement.

Syntax

- RUN procedurename. //Runs a procedure called procedurename.
- RUN proc1(INPUT "HELLO"). //Inputs the string HELLO to proc1
- RUN proc2(INPUT var1, output var2). //Inputs var1 and outputs var2 to/from proc2
- RUN proc3(input "name = 'joe'", OUTPUT TABLE ttResult). //Inputs name=joe and outputs records in a table
- PROCEDURE proc: // Declares a procedure named proc
- END PROCEDURE. // Ends the current procedure

Examples

A basic internal procedure

Unlike functions, there's no need to forward declare a procedure. It can be placed anywhere in your code, before or after you call it using run.

```
RUN proc.

//Procedure starts here
PROCEDURE proc:

//Procedure ends here
END PROCEDURE.
```

The procedure name is folowed by a colon sign telling us that this is the start of a block. The block ends with END PROCEDURE. (but this can be replaced with simply END.).

INPUT and OUTPUT parameters
A procedure can have parameters of different kinds: input, output, input-output (bidirectional) and also some special types like temp-tables and datasets).

In the run statement it's optional to declare `INPUT` (it's considered default) - all other directions must be specifically declared.

A procedure taking two integers as input and outputting a decimal.

```plaintext
PROCEDURE divideAbyB:
   DEFINE INPUT PARAMETER piA AS INTEGER NO-UNDO.
   DEFINE INPUT PARAMETER piB AS INTEGER NO-UNDO.
   DEFINE OUTPUT PARAMETER pdeResult AS DECIMAL NO-UNDO.

   pdeResult = piA / piB.
END PROCEDURE.

DEFINE VARIABLE de AS DECIMAL NO-UNDO.
RUN divideAbyB(10, 2, OUTPUT de).
DISPLAY de. //5.00
```

Parameters are totally optional. You can mix and match any way you want. The order of the parameters are up to you but it's handy to start with input and end with output - you have to put them in the right order in the run statement and mixing directions can be annoying.

**Recursion - see recursion**

Recursion is easy - run the procedure itself from inside the procedure. However if you recurse too far the stack will run out of space.

A procedure calculation the factorial.

```plaintext
PROCEDURE factorial:
   DEFINE INPUT PARAMETER piNum AS INTEGER NO-UNDO.
   DEFINE OUTPUT PARAMETER piFac AS INTEGER NO-UNDO.

   DEFINE VARIABLE iFac AS INTEGER NO-UNDO.
   IF piNum = 1 THEN DO:
      pifac = 1.
   END.
   ELSE DO:
      RUN factorial(piNum - 1, OUTPUT iFac).
      piFac = piNum * iFac.
   END.
END PROCEDURE.

DEFINE VARIABLE f AS INTEGER NO-UNDO.
RUN factorial(7, OUTPUT f).
DISPLAY f.
```
Scope

The procedure has its own scope. The outside scope will "bleed" into the procedure but not the other way around.

```
DEFINE VARIABLE i AS INTEGER     NO-UNDO INIT 1.
DEFINE VARIABLE j AS INTEGER     NO-UNDO.

PROCEDURE p:
    MESSAGE i VIEW-AS ALERT-BOX. // 1
    MESSAGE j VIEW-AS ALERT-BOX. // 0

j = 2.
END PROCEDURE.

RUN p.

MESSAGE i VIEW-AS ALERT-BOX. // 1
MESSAGE j VIEW-AS ALERT-BOX. // 2
```

Declaring a variable inside a procedure that has the same name as a parameter on the outside will only create a local variable.

```
DEFINE VARIABLE i AS INTEGER     NO-UNDO INIT 1.
DEFINE VARIABLE j AS INTEGER     NO-UNDO.

PROCEDURE p:
    DEFINE VARIABLE i AS INTEGER     NO-UNDO INIT 5.

    MESSAGE i VIEW-AS ALERT-BOX. // 5
    MESSAGE j VIEW-AS ALERT-BOX. // 0

    j = 2.
END PROCEDURE.

RUN p.

MESSAGE i VIEW-AS ALERT-BOX. // 1
MESSAGE j VIEW-AS ALERT-BOX. // 2
```

Any variable created on the inside of a procedure is accessible to that procedure only.

This will generate a compiler error:

```
PROCEDURE p:

    DEFINE VARIABLE i AS INTEGER     NO-UNDO INIT 5.

END PROCEDURE.

RUN p.
```
MESSAGE i VIEW-AS ALERT-BOX. // Unknown Field or Variable name i - error 201

Read Procedures online: https://riptutorial.com/progress-4gl/topic/8914/procedures
Chapter 9: Queries

Introduction

The examples will be based on a copy of the demo database *Sports 2000* provided with the setup of Progress.

When working with queries in Progress you need to:

- **DEFINE** the query and set what buffers (tables) and fields it works against.
- **OPEN** the query with a specific *WHERE*-clause that defines how to retrieve the records. Possibly also sorting (*BY/BREAK BY*)
- **GET** the actual data - that can be the `FIRST`, `NEXT`, `PREV` (for previous) or `LAST` matching record.

Syntax

- **DEFINE QUERY** `query-name` FOR `buffer-name` . //General query definition for one buffer
- **DEFINE QUERY** `query-name` FOR `buffer-name1, buffer-name2` . //Joining two buffers
- **DEFINE QUERY** `query-name` FOR `buffer-name` FIELDS (field1 field2). //Only retreive field1 and field2
- **DEFINE QUERY** `query-name` FOR `buffer-name` EXCEPT (field3). //Retreive all fields except field3.

Examples

Basic Query

```plaintext
/* Define a query named q1 for the Customer table */
DEFINE QUERY q1 FOR Customer.
/* Open the query for all Customer records where the state is "tx" */
OPEN QUERY q1 FOR EACH Customer WHERE Customer.state = 'TX'.

/* Get the first result of query q1 */
GET FIRST q1.

/* Repeat as long as query q1 has a result */
DO WHILE NOT QUERY-OFF-END('q1'):
    /* Display Customer.Name in a frame called frame1 with 10 rows */
    DISPLAY Customer.Name WITH FRAME frame1 10 DOWN.
    /* Move down the target line where to display the next record */
    DOWN WITH FRAME frame1.
    /* Get the next result of query q1 */
    GET NEXT q1.
END.
/* Display how many results query q1 had. */
DISPLAY NUM-RESULTS('q1') LABEL "Number of records".
```

https://riptutorial.com/
Multi-Tables Query

This query will join three tables: Customer, Order and Orderline.

The use of the OF statement as in `childtable OF parenttable` assumes that indexes are constructed in a specific way. That is the case in the sports2000-database.

```
DEFINE QUERY q1 FOR Customer, Order, Orderline.
OPEN QUERY q1 FOR EACH Customer WHERE Customer.state = 'TX', EACH Order OF customer WHERE order.custnum < 1000, EACH orderline OF order.
GET FIRST q1.
DO WHILE NOT QUERY-OFF-END('q1'):
    DISPLAY Customer.Name Order.OrderNum OrderLine.LineNum WITH FRAME frameA 20 DOWN.
```
RESULT: In Windows GUI:

<table>
<thead>
<tr>
<th>Name</th>
<th>Order Num</th>
<th>Line Num</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thundering Surf Inc.</td>
<td>23</td>
<td>1</td>
</tr>
<tr>
<td>Thundering Surf Inc.</td>
<td>23</td>
<td>2</td>
</tr>
<tr>
<td>Thundering Surf Inc.</td>
<td>41</td>
<td>1</td>
</tr>
<tr>
<td>Thundering Surf Inc.</td>
<td>41</td>
<td>2</td>
</tr>
<tr>
<td>Thundering Surf Inc.</td>
<td>41</td>
<td>3</td>
</tr>
<tr>
<td>Thundering Surf Inc.</td>
<td>134</td>
<td>1</td>
</tr>
<tr>
<td>Thundering Surf Inc.</td>
<td>134</td>
<td>2</td>
</tr>
<tr>
<td>Thundering Surf Inc.</td>
<td>134</td>
<td>3</td>
</tr>
<tr>
<td>Thundering Surf Inc.</td>
<td>134</td>
<td>4</td>
</tr>
<tr>
<td>Thundering Surf Inc.</td>
<td>134</td>
<td>5</td>
</tr>
</tbody>
</table>

Procedure complete. Press space bar to continue.

Moving position within a query using next, first, prev and last

DEFINE QUERY q1 FOR Customer.

OPEN QUERY q1 FOR EACH Customer.

GET FIRST q1.

loop:
REPEAT:
   IF AVAILABLE Customer THEN DO:
      DISPLAY Customer.NAME CustNum WITH FRAME frClient TITLE "Client data".
      DISPLAY "(P)revious" SKIP
      "(N)ext" SKIP

"(F)irst" SKIP
"(L)ast" SKIP
"(Q)uit" SKIP
WITH FRAME frInstr
    TITLE "Instructions".
END.

READKEY.

IF LASTKEY = ASC("q") THEN LEAVE loop.
ELSE IF LASTKEY = ASC("n") THEN
    GET NEXT q1.
ELSE IF LASTKEY = ASC("p") THEN
    GET PREV q1.
ELSE IF LASTKEY = ASC("l") THEN
    GET LAST q1.
ELSE IF LASTKEY = ASC("f") THEN
    GET FIRST q1.
END.

MESSAGE "Bye" VIEW-AS ALERT-BOX.

Read Queries online: https://riptutorial.com/progress-4gl/topic/8694/queries
Chapter 10: Strings

Introduction

In Progress ABL there are two types of strings, those defined as CHARACTER and those defined as LONGCHAR. A file larger than 32K in length is a LONGCHAR. Most strings are unless specified any other way case insensitive.

Remarks

Remember - all positions start with the position 1!

Examples

Defining, assing and displaying a string

Generally you should always define all variable and parameters as NO-UNDO unless you really need to.

```abl
DEFINE VARIABLE cString AS CHARACTER NO-UNDO.
cString = "HELLO".
DISPLAY cString.
```

Concatenating strings

Using the + operator you can easily concatenate two or more strings.

```abl
DEFINE VARIABLE cString AS CHARACTER NO-UNDO.
cString = "HELLO".
cString = cString + " " + "GOODBYE".
DISPLAY cString FORMAT "X(20)".
```

String manipulation

There are a couple of useful built in functions for working with string. All functions working with the position of characters start with index 1 as the first character, not 0 as is common in many languages.

**STRING** - converts any value to a string

This example converts the integer 2000 to the string "2000".

https://riptutorial.com/
DEFINE VARIABLE i AS INTEGER NO-UNDO.
DEFINE VARIABLE c AS CHARACTER NO-UNDO.
i = 2000.
c = STRING(i).
DISPLAY c.

CHR and ASC - converts single characters to and from ascii.

CHR(integer)
Returns the character representation for ascii code integer

ASC(character)
Returns the ascii integer value for the character

DEFINE VARIABLE ix AS INTEGER NO-UNDO.
DEFINE VARIABLE letter AS CHARACTER NO-UNDO FORMAT "X(1)" EXTENT 26.
DO ix = 1 TO 26:
  letter[ix] = CHR((ASC("A")) - 1 + ix).
END.
DISPLAY SKIP(1) letter WITH 2 COLUMNS NO-LABELS
  TITLE "T H E  A L P H A B E T".

LENGTH - returns the length of a string

LENGTH(string). //Returns an integer with the length of the string.

DEFINE VARIABLE cString AS CHARACTER NO-UNDO.
cString = "HELLO".
MESSAGE "The string " cString " is " LENGTH(cString) " characters long" VIEW-AS ALERT-BOX.

SUBSTRING - returns or assigns a part of a string

  • SUBSTRING(string, starting-position, length).

Returns "length" characters from "string" starting on position "starting-position".

  • SUBSTRING(string, starting-position).

Returns the rest of "string", starting at position "starting-position"

DEFINE VARIABLE cString AS CHARACTER NO-UNDO.
cString = "ABCDEFGH".
DISPLAY SUBSTRING(cString, 4, 2). //Displays "DE"
DISPLAY SUBSTRING(cString, 4). //Displays "DEFGH"

Substring can also be used to overwrite a part of a string. Use the same syntax but assign that substring instead:

DEFINE VARIABLE cString AS CHARACTER NO-UNDO.
cString = "ABCDEFGH".
SUBSTRING(cString, 4, 2) = "XY". //Replaces position 4 and 5 with "XY"
DISPLAY cString.

There’s also a similar function called OVERLAY this example from the Progress documentation covers the differences between OVERLAY and SUBSTRING:

/* This procedure illustrates the differences between the SUBSTRING and OVERLAY statements. */
DEFINE VARIABLE cOriginal AS CHARACTER NO-UNDO INITIAL "OpenEdge".
DEFINE VARIABLE cSubstring AS CHARACTER NO-UNDO.
DEFINE VARIABLE cOverlay AS CHARACTER NO-UNDO.
DEFINE VARIABLE cResults AS CHARACTER NO-UNDO.

/* Default behavior without optional LENGTH. */
ASSIGN
cSubstring = cOriginal
SUBSTRING(cSubstring, 2) = "***"
cOverlay = cOriginal
OVERLAY(cOverlay, 2) = "***"
cResults = "target = ~"OpenEdge~". ~n~n" + "If you do not supply a length, SUBSTRING and OVERLAY default as follows: ~n~n" + "SUBSTRING(target,2) = ~"***~" yields: " + cSubstring + ". ~n"
+ "OVERLAY(target,2) = ~"***~" yields: " + cOverlay + ".".

/* Behavior with zero LENGTH. */
ASSIGN
cSubstring = cOriginal
SUBSTRING(cSubstring, 2, 0) = "***"
cOverlay = cOriginal
OVERLAY(cOverlay, 2, 0) = "***"
cResults = cResults + ~n~n" + "For a zero length, SUBSTRING and OVERLAY behave as follows: ~n~n"
+ "SUBSTRING(target,2) = ~"***~" yields: " + cSubstring + ". ~n"
+ "OVERLAY(target,2) = ~"***~" yields: " + cOverlay + ".".

/* Behavior with LENGTH < replacement. */
ASSIGN
cSubstring = cOriginal
SUBSTRING(cSubstring, 2, 1) = "***"
cOverlay = cOriginal
OVERLAY(cOverlay, 2, 1) = "***"
cResults = cResults + ~n~n" + "For a length shorter than the replacement, SUBSTRING and OVERLAY behave as follows: ~n~n"
+ "SUBSTRING(target,2,1) = ~"***~" yields: " + cSubstring + ". ~n"
+ "OVERLAY(target,2,1) = ~"***~" yields: " + cOverlay + ".".

/* Behavior with LENGTH = replacement. */
**INDEX** - return the position of a string in a string.

**INDEX(source, target)**

Search target within source (left to right) and return its position. If it's missing return 0.

**INDEX(source, target, starting-position).**

Same as above but start searching at starting-position

```plaintext
DEFINE VARIABLE str AS CHARACTER NO-UNDO.
str = "ABCDEFGH".
DISPLAY INDEX(str, "cd") INDEX(str, "cd", 4). //Will display 3 and 0
```

**REPLACE** - replaces a string within a string.

**REPLACE(string, from-string, to-string)**

Replaces from-string with to-string in string. From-string and to-string don't need to be of the same length, to-string can also be nothing (""") to remove a character.

```plaintext
DEFINE VARIABLE c AS CHARACTER NO-UNDO.
c = "ELLO".
DISPLAY REPLACE(c, "E", "HE"). // Displays "HELLO"
```
c = "ABABABA".

DISPLAY REPLACE(c, "B", ""). // Remove all Bs

**TRIM** - removes leading and trailing whitespaces (or other characters).

This can be useful when cleaning up indata.

**TRIM**(string)

Removes all leading and trailing spaces, tabs, line feeds, carriage returns.

**TRIM**(string, character).

Removes all leading and trailing "characters".

**LEFT-TRIM** and **RIGHT-TRIM** does the same thing but only leading or trailing.

---

**DEFINE VARIABLE** c **AS CHARACTER** **NO-UNDO**.

c = "__HELLO_WORLD_____".

DISPLAY TRIM(c, "_").

/*Displays HELLO_WORLD without all the leading and trailing underscores but leaves the one in the middle. REPLACE would have removed that one as well */

**SUBSTITUTE** - substitutes parameters in a string.

**SUBSTITUTE** is a limited function for replacing up to nine preformatted parameters in a string.

**SUBSTITUTE**(string, param1, param2, ..., param9).

The parameters must be in the format &1 to &9.

If you want to use an ampersand in the string (and not use it as a parameter) escape it with another ampersand: &&.

---

**DEFINE VARIABLE** str **AS CHARACTER** **NO-UNDO**.

str = "&1 made &2 goals in &3 games playing for &4".

MESSAGE SUBSTITUTE(str, "Zlatan Ibrahimovic", 113, 122, "Paris Saint-Germain") VIEW-AS ALERT-BOX.

MESSAGE SUBSTITUTE(str, "Mats Sundin", 555, 1305, "Toronto Maple Leafs") VIEW-AS ALERT-BOX.

A parameter can appear more than once in a string, all will be replaced:

MESSAGE SUBSTITUTE("&1 &2 or not &1 &2", "To", "Be") VIEW-AS ALERT-BOX.

**CASE-SENSITIVE** strings
All strings in Progress ABL are case sensitive unless specified otherwise.

This example will display a message box saying that the strings are identical.

```plaintext
DEFINE VARIABLE str1 AS CHARACTER   NO-UNDO.
DEFINE VARIABLE str2 AS CHARACTER   NO-UNDO.
str1 = "abc".
str2 = "ABC".
IF str1 = str2 THEN
    MESSAGE "The strings are identical" VIEW-AS ALERT-BOX.
```

To declare a string case sensitive you just add the attribute `CASE-SENSITIVE`.

```plaintext
DEFINE VARIABLE str1 AS CHARACTER   NO-UNDO CASE-SENSITIVE.
DEFINE VARIABLE str2 AS CHARACTER   NO-UNDO.
str1 = "abc".
str2 = "ABC".
IF str1 = str2 THEN
    MESSAGE "The strings are identical" VIEW-AS ALERT-BOX.
ELSE
    MESSAGE "There's a difference" VIEW-AS ALERT-BOX.
```

(It's enough that one of the strings has it in this case).

**BEGINS and MATCHES**

**BEGINS** - returns TRUE if one string begins with another string.

```
string1 BEGINS string2
```

If string1 BEGINS with (or is equal to) string2 this will return true. Otherwise it will return false. If string two is empty ("") it will always return true.

**BEGINS** is very useful in queries where you want to search the beginning of something, for instance a name. But it's basically a function working on strings.

```plaintext
DEFINE VARIABLE str AS CHARACTER   NO-UNDO.
DEFINE VARIABLE beg AS CHARACTER   NO-UNDO.
str = "HELLO".
beg = "HELLO".
DISPLAY str BEGINS beg. // yes

str = "HELLO".
beg = "H".
DISPLAY str BEGINS beg. // yes

str = "HELLO".
beg = "".
DISPLAY str BEGINS beg. // yes
```
MATCHES returns true if certain wildcard criteria is met in a string.

string1 MATCHES expression

Returns true if string1 matches the wildcard expression:

* (asterisk) = 0 to n characters (basically any string of any length)

. (period) = wildcard for any character (except null)

Converting upper and lower case

As mentioned before strings are normally case insensitive but that only regards comparison of strings. There’s built in functions for changing case.

CAPS (string)

Makes string upper case

LC(string)

Makes string lower case
DEFINE VARIABLE c AS CHARACTER  NO-UNDO.
DEFINE VARIABLE d AS CHARACTER  NO-UNDO.
c = "Hello".
d = "World".
DISPLAY CAPS(c) LC(d). // HELLO world

Remember strings normally are case insensitive

DEFINE VARIABLE c AS CHARACTER  NO-UNDO.
DEFINE VARIABLE d AS CHARACTER  NO-UNDO.
c = "hello".
d = "hello".
DISPLAY CAPS(c) = LC(d). // yes

Unless specified as CASE-SENSITIVE

DEFINE VARIABLE c AS CHARACTER  NO-UNDO CASE-SENSITIVE.
DEFINE VARIABLE d AS CHARACTER  NO-UNDO.
c = "hello".
d = "hello".
DISPLAY CAPS(c) = LC(d). // no

Lists

There are a number of functions and methods for working with comma (or other character) separated lists in Progress 4GL.

**NUM-ENTRIES** Returns the number of entries in a list. You can optionally specify delimiter, comma is default

NUM-ENTRIES(string [, delimiter])

Using comma, the default delimiter:

DEFINE VARIABLE cList AS CHARACTER  NO-UNDO.
cList = "Goodbye,cruel,world!".
DISPLAY NUM-ENTRIES(cList). //3

Using another delimiter, semicolon:

DEFINE VARIABLE cList AS CHARACTER  NO-UNDO.
cList = "Goodbye;cruel;world!".
DISPLAY NUM-ENTRIES(cList, ";"). //3

https://riptutorial.com/
ENTRY - function - returns a specified entry in a list

As usual starting position is 1, not 0!

ENTRY( entry, list [, delimiter]).

```
DEFINE VARIABLE cList AS CHARACTER   NO-UNDO.
cList = "Goodbye,cruel,world!".
DISPLAY ENTRY(2, cList). //cruel
```

ENTRY - method - assigning the value of a specified entry in a list

ENTRY( entry, list [, delimiter]) = value

```
DEFINE VARIABLE cList AS CHARACTER   NO-UNDO.
cList = "Goodbye,cruel,world!".
ENTRY(1, cList) = "Hello".
ENTRY(2, cList) = "nice".
MESSAGE REPLACE(cList, ",", " ") VIEW-AS ALERT-BOX. //Hello nice world!
```

LOOKUP - check a list for a specific entry. Returns it's entry.

If the string isn't present in the list lookup will returns 0

LOOKUP(string, list [, delimiter])

```
DEFINE VARIABLE cList AS CHARACTER   NO-UNDO.
cList = "Hello,nice,world!".
MESSAGE LOOKUP("nice", cList) VIEW-AS ALERT-BOX. //2
MESSAGE LOOKUP("cruel", cList) VIEW-AS ALERT-BOX. //0
```

Special characters (and escaping)

In Progress 4GL the normal way to write a special character is to preceed it with a tilde character (~).

These are the default special characters

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Interpreted as</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>~&quot;</td>
<td>&quot;</td>
<td>Used to write &quot; inside strings defined using &quot;string&quot;.</td>
</tr>
<tr>
<td>~'</td>
<td>'</td>
<td>Used to write ' inside strings defined using 'string'.</td>
</tr>
<tr>
<td>~~</td>
<td>~</td>
<td>For instance if you want to print the sequence and not how</td>
</tr>
</tbody>
</table>

https://riptutorial.com/
<table>
<thead>
<tr>
<th>Sequence</th>
<th>Interpreted as</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>~\</td>
<td>\</td>
<td></td>
</tr>
<tr>
<td>~{</td>
<td>{</td>
<td>{ is used in preprocessors and sometimes escaping is needed.</td>
</tr>
<tr>
<td>~nnn</td>
<td>A single character</td>
<td>nnn is an octal number representing the ascii value of the character.</td>
</tr>
<tr>
<td>~t</td>
<td>tab</td>
<td></td>
</tr>
<tr>
<td>~n</td>
<td>New line/line feed</td>
<td></td>
</tr>
<tr>
<td>~r</td>
<td>Carriage return</td>
<td></td>
</tr>
<tr>
<td>~E</td>
<td>Escape</td>
<td></td>
</tr>
<tr>
<td>~b</td>
<td>Backspace</td>
<td></td>
</tr>
<tr>
<td>~f</td>
<td>Form feed</td>
<td></td>
</tr>
</tbody>
</table>

If you want to display tilde at all it must be escaped!

```plaintext
MESSAGE "A single tilde: ~" VIEW-AS ALERT-BOX.
MESSAGE "At sign: ~100" SKIP
   "Tab-separated-words!" SKIP
   "A linefeed:~n"
   "Escaping a quote sign: ~"This is a quote!~"" SKIP VIEW-AS ALERT-BOX.
```

Read Strings online: [https://riptutorial.com/progress-4gl/topic/8872/strings](https://riptutorial.com/progress-4gl/topic/8872/strings)
Chapter 11: TEMP-TABLE

Introduction

The **TEMP-TABLE** is a very powerful feature of Progress ABL. It's a temporary in-memory (mostly at least) table that can be used for writing complex logic. It can be used as input/output parameters to procedures, functions and other programs. One or more temp-tables can make up the foundation of a **DATASET** (often called ProDataset).

Almost anything that can be done with a native Progress database table can be done with a temp-table.

Examples

Defining a simple temp-table

This is the definition of a **TEMP-TABLE** named ttTempTable with three fields. **NO-UNDO** indicates that no undo handling is needed (this is usually what you want to do unless you really need the opposite).

```
DEFINE TEMP-TABLE ttTempTable NO-UNDO
  FIELD field1 AS INTEGER
  FIELD field2 AS CHARACTER
  FIELD field3 AS LOGICAL.
```

A temp-table with an index

Temp-tables can (and should) be created with indices if you plan to run queries against them.

This table has one index (index1) containing of one field (field1). This index is primary and unique (meaning not two records can have the same contents of field1).

```
DEFINE TEMP-TABLE ttTempTable NO-UNDO
  FIELD field1 AS INTEGER
  FIELD field2 AS CHARACTER
  FIELD field3 AS LOGICAL
  INDEX index1 IS PRIMARY UNIQUE field1.
```

More indexes - indices...

You can define multiple indices for each temp-table. If you need them - define them. Basically an index matching your query and/or sort order will help performance!

```
DEFINE TEMP-TABLE ttWithIndex NO-UNDO
  FIELD field1 AS INTEGER
  FIELD field2 AS CHARACTER
  FIELD field3 AS LOGICAL
  INDEX field1 field1.
```
DEFINE TEMP-TABLE ttWithoutIndex NO-UNDO
    FIELD field1 AS INTEGER
    FIELD field2 AS CHARACTER
    FIELD field3 AS LOGICAL.

DEFINE VARIABLE i       AS INTEGER     NO-UNDO.
DEFINE VARIABLE iWithCreate AS INTEGER     NO-UNDO.
DEFINE VARIABLE iWithFind  AS INTEGER     NO-UNDO.
DEFINE VARIABLE iWithoutCreate AS INTEGER     NO-UNDO.
DEFINE VARIABLE iWithoutFind AS INTEGER     NO-UNDO.

ETIME(TRUE).
DO i = 1 TO 1000:
    CREATE ttWithIndex.
    ttWithIndex.field1 = i.
END.
iWithCreate = ETIME.

ETIME(TRUE).
DO i = 1 TO 1000:
    CREATE ttWithoutIndex.
    ttWithoutIndex.field1 = i.
END.
iWithoutCreate = ETIME.

RELEASE ttWithIndex.
RELEASE ttWithoutIndex.

ETIME(TRUE).
DO i = 1 TO 1000:
    FIND FIRST ttWithIndex WHERE ttWithIndex.field1 = i NO-ERROR.
END.
iWithFind = ETIME.

ETIME(TRUE).
DO i = 1 TO 1000:
    FIND FIRST ttWithoutIndex WHERE ttWithoutIndex.field1 = i NO-ERROR.
END.
iWithoutFind = ETIME.

MESSAGE
"With index took" iWithFind "ms to find and" iWithCreate "ms to create" SKIP
"Without index took" iWithoutFind "ms to find and" iWithoutCreate "ms to create"
VIEW-AS ALERT-BOX.

With index took 7 ms to find and 7 ms to create
Without index took 480 ms to find and 9 ms to create

Searching with index was roughly 70 times faster compared to no index! This is just one run of course so not a scientific proof but your index setup will make impact.
Inputting and outputting temp-tables

It's very simple to pass temp-tables in and out of programs, procedures and functions.

This can be handy if you want a procedure to process a bigger number of data than you can easily store in a string or similar. You can pass temp-tables as INPUT, OUTPUT and INPUT-OUTPUT data.

Inputting one temp-table and outputting another:

```plaintext
DEFINE TEMP-TABLE ttRequest NO-UNDO
  FIELD fieldA AS CHARACTER
  FIELD fieldB AS CHARACTER.

/* Define a temp-table with the same fields and indices */
DEFINE TEMP-TABLE ttResponse NO-UNDO LIKE ttRequest.

/* A procedure that simply swap the values of fieldA and fieldB */
PROCEDURE swapFields:
  DEFINE INPUT PARAMETER TABLE FOR ttRequest.
  DEFINE OUTPUT PARAMETER TABLE FOR ttResponse.
  FOR EACH ttRequest:
    CREATE ttResponse.
    ASSIGN
      ttResponse.fieldA = ttRequest.fieldB
      ttResponse.fieldB = ttRequest.fieldA.
    END.
  END PROCEDURE.

CREATE ttRequest.
ASSIGN ttRequest.fieldA = "A"
  ttRequest.fieldB = "B".
CREATE ttRequest.
ASSIGN ttRequest.fieldA = "B"
  ttRequest.fieldB = "C".
CREATE ttRequest.
ASSIGN ttRequest.fieldA = "C"
  ttRequest.fieldB = "D".

/* Call the procedure */
RUN swapFields ( INPUT  TABLE ttRequest
  , OUTPUT TABLE ttResponse).

FOR EACH ttResponse:
  DISPLAY ttResponse.
END.
```

Result:

```
fieldA--------fieldB--------
  B             A
  C             B
  D             C
```
**Input-outputting a temp-table:**

```plaintext
DEFINE TEMP-TABLE ttCalculate NO-UNDO
    FIELD num1 AS INTEGER
    FIELD num2 AS INTEGER
    FIELD response AS DECIMAL.

PROCEDURE pythagoras:
    DEFINE INPUT-OUTPUT PARAMETER TABLE FOR ttCalculate.
    FOR EACH ttCalculate:
        ttCalculate.response = SQRT( EXP(num1, 2) + EXP(num2, 2)).
    END.
END PROCEDURE.

CREATE ttCalculate.
ASSIGN ttCalculate.num1 = 3
ASSIGN ttCalculate.num2 = 4.
CREATE ttCalculate.
ASSIGN ttCalculate.num1 = 6
ASSIGN ttCalculate.num2 = 8.
CREATE ttCalculate.
ASSIGN ttCalculate.num1 = 12
ASSIGN ttCalculate.num2 = 16.

/* Call the procedure */
RUN pythagoras ( INPUT-OUTPUT TABLE ttCalculate ).
FOR EACH ttCalculate:
    DISPLAY ttCalculate.
END.

Result:

```
<table>
<thead>
<tr>
<th>num1</th>
<th>num2</th>
<th>response</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>5.00</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>10.00</td>
</tr>
<tr>
<td>12</td>
<td>16</td>
<td>20.00</td>
</tr>
</tbody>
</table>
```

**Passing to functions**

```plaintext
DEFINE TEMP-TABLE ttNumbers NO-UNDO
    FIELD num1 AS INTEGER
    FIELD num2 AS INTEGER
    INDEX index1 num1 num2.

DEFINE VARIABLE iNum AS INTEGER NO-UNDO.

/* Forward declare the function */
FUNCTION hasAPair RETURNS LOGICAL (INPUT TABLE ttNumbers) FORWARD.

DO iNum = 1 TO 100:
    CREATE ttNumbers.
    ASSIGN ttNumbers.num1 = RANDOM(1,100)
```
**Passing to program files**

You pass temp-tables to and from other .p-programs the same way you pass them to other procedures. The only difference is that both the calling and the called program must have the same temp-table declaration. One easy way is to store the temp-table program in a third file - an include that's used in both programs.

Include file containing temp-table definition: /* ttFile.i */

```
DEFINE TEMP-TABLE ttFile NO-UNDO
FIELD fName AS CHARACTER FORMAT "x(20)"
FIELD isADirectory AS LOGICAL.
```

Program checking all files in a temp-table. Are they directories?

```
/* checkFiles.p */
{ttFile.i}
DEFINE INPUT-OUTPUT PARAMETER TABLE FOR ttFile.

FOR EACH ttFile:
    FILE-INFO:FILE-NAME = ttFile.fName.
    IF FILE-INFO:FILE-TYPE BEGINS "D" THEN
        ttFile.isADirectory = TRUE.
    END.
END.
```

Main program:

```
{ttFile.i}

CREATE ttFile.
ASSIGN ttFile.fname = "c:\temp\".

CREATE ttFile.
ASSIGN ttFile.fname = "c:\Windows\".

CREATE ttFile.
ASSIGN ttFile.fname = "c:\Windoose\".

RUN checkFiles.p(INPUT-OUTPUT TABLE ttFile).
```
FOR EACH ttFile:
    DISPLAY ttFile.
END.

Result:

<table>
<thead>
<tr>
<th>fName</th>
<th>isADirector</th>
</tr>
</thead>
<tbody>
<tr>
<td>c:\temp\</td>
<td>yes</td>
</tr>
<tr>
<td>c:\Windows\</td>
<td>yes</td>
</tr>
<tr>
<td>c:\Windoose\</td>
<td>no</td>
</tr>
</tbody>
</table>

Read TEMP-TABLE online: https://riptutorial.com/progress-4gl/topic/8957/temp-table
Chapter 12: Variables

Introduction

Progress ABL is statically typed. The variables need to be declared and the datatype cannot be changed during run time.

Syntax

- DEFINE VARIABLE i AS INT64 INITIAL -200 NO-UNDO. //A 64-bit integer initialized to -200
- DEFINE VARIABLE l AS LOGICAL NO-UNDO. //A logical variable named l
- DEFINE VARIABLE c AS CHARACTER NO-UNDO CASE-SENSITIVE. //A case sensitive ('a' <> 'A') variable.
- DEFINE VARIABLE dt AS DATE INITIAL TODAY NO-UNDO. //A date variable set to todays date.
- DEFINE VARIABLE a AS CHARACTER EXTENT 5 NO-UNDO. //An character array with length = 5
- DEFINE VARIABLE j AS INTEGER EXTENT NO-UNDO. //An extent without a set length
- DEFINE VARIABLE b AS DATETIME LABEL "Departure time". //A variable with a label

Examples

Basic variable declarations

/*
These variables are declared with 'NO-UNDO'.
That states that no undo handling is wanted for this specific variable in case of a transactional roll-back.
This should always be the default unless transactional control over this variable is a requirement.
*/

/* Strings. A character longer than 32K should be a longchar */
DEFINE VARIABLE c AS CHARACTER NO-UNDO.
DEFINE VARIABLE cl AS LONGCHAR NO-UNDO.

/* Integers and decimals. INTEGER = 32 bit. INT64 = 64 bits */
DEFINE VARIABLE i AS INTEGER NO-UNDO.
DEFINE VARIABLE j AS INT64 NO-UNDO.
DEFINE VARIABLE k AS DECIMAL NO-UNDO.

/* Date and datetimes. Unset variables have the unknown value ? */
DEFINE VARIABLE d AS DATE NO-UNDO.
DEFINE VARIABLE dt AS DATETIME NO-UNDO.
DEFINE VARIABLE dtz AS DATETIME-TZ NO-UNDO.

/* LOGICAL = Boolean data. True or false (or ?) */
DEFINE VARIABLE l AS LOGICAL NO-UNDO.

/* Rowids and recids are internal identifiers to database records */
DEFINE VARIABLE rid AS ROWID NO-UNDO.
DEFINE VARIABLE rec AS RECID NO-UNDO.

/* A handle is a handle to anything: a session, an on screen widget etc */
/* A Com-handle is used for ActiveX Com-automation */
DEFINE VARIABLE h AS HANDLE NO-UNDO.
DEFINE VARIABLE hc AS COM-HANDLE NO-UNDO.

/* A raw variable can contain any data. Binary, strings etc */
DEFINE VARIABLE rw AS RAW NO-UNDO.

/* A mempointer contains a sequence of bytes in memory. */
DEFINE VARIABLE m AS MEMPTR NO-UNDO.

Arrays - defining and accessing

Progress supports one dimensional arrays, but they are called extents.

/* Define a character array with the length 5, and display it's length */
DEFINE VARIABLE a AS CHARACTER EXTENT 5 NO-UNDO.
DISPLAY EXTENT(a).

Individual positions in the array is accessed using "standard" c-style brackets. But the index starts at 1. The maximum size is 28000.

a[1] = "A".
a[2] = "B".
a[3] = "C".
a[4] = "D".
a[5] = "E".
DISPLAY a[5].

Result:

Index 0 will generate an error:

DISPLAY a[0].

Result:
You can also define an indeterminate array without a set length. The length (extent) can be set in run-time. But only once!

```
DEFINE VARIABLE a AS CHARACTER EXTENT NO-UNDO.
EXTENT(a) = 10.
EXTENT(a) = 1.
```

The third line will produce the following error:

```
** Array subscript is less than 1 or greater than extent. (367)
** Could not understand line 9. (190)
```

You can use the `INITIAL` option on the `DEFINE VARIABLE` statement to set initial values.

```
DEFINE VARIABLE a AS CHARACTER EXTENT 3 INITIAL ["one","two","three"] NO-UNDO.
/* Some statements (like DISPLAY) can handle a whole array: */
DISPLAY a.
```

Result:

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>a2</td>
<td>a3</td>
</tr>
<tr>
<td>one</td>
<td>two</td>
<td>three</td>
</tr>
</tbody>
</table>
```

If you don’t set all extents the remaining will get the last set value:

```
DEFINE VARIABLE a AS CHARACTER EXTENT 10 INITIAL ["one","two","three"] NO-UNDO.
DISPLAY a.
```

Result:
Using the LIKE keyword

Using `LIKE` you can base the definition of your variable on another variable or a field in a database or temp-table.

Defining a variable `LIKE` a database field requires the database to always be connected. This might not always be what you want.

```plaintext
DEFINE VARIABLE i AS INTEGER NO-UNDO LABEL "Nr" FORMAT "99999". /* Define a variable with the same properties as "i" */
DEFINE VARIABLE j LIKE i.

/* Define a variable based on Customer.Custnum from the sports2000 database but override the label-definition */
DEFINE VARIABLE k LIKE Customer.Custnum LABEL "Client".
```

Read Variables online: https://riptutorial.com/progress-4gl/topic/8800/variables
Chapter 13: Working with numbers

Introduction

Progress ABL supports three number formats: 32 and 64 bit integers and floats.

Examples

Operators

Progress supports + / - * as operators. They cannot be overloaded. Division always returns a decimal. If any of the numbers in a calculation is a decimal a decimal will be returned. Otherwise an integer or int64.

There's no += or ++ operator. To increase or decrease a variable you have to assign it to itself plus or minus something. So to add 1 to a variable you do: i = i + 1.

```
DEFINE VARIABLE i AS INTEGER     NO-UNDO.
DEFINE VARIABLE j AS INTEGER     NO-UNDO.

i = 3.
j = 2.

DISPLAY i + j. // 3 + 2 = 5
DISPLAY i - j. // 3 - 2 = 1
DISPLAY i / j. // 3 / 2 = 1.5
DISPLAY INTEGER(i / j). //Integer(3/2) = 2.
DISPLAY i * j. //3 x 2 = 6
```

More mathematical functions

**EXP** - Returns the result of raising a number to a power.

```
EXP( base, exponent)
```

```
MESSAGE EXP(10, 2) VIEW-AS ALERT-BOX. // Messages 100
```

**SQRT** - Returns the square root of a number.

```
SQRT( number)
```

```
MESSAGE "The square root of 256 is " SQRT(256) VIEW-AS ALERT-BOX. // Messages 16
```

**MODULO** - Determines the remainder after division.
expression MODULO base

DISPLAY 52 MODULO 12. //Displays 4

ROUND - Rounds a decimal expression to a specified number of places after the decimal point.

ROUND( number, precision)

DISPLAY ROUND(67.12345, 6) FORMAT "99.99999". // 67.12345
DISPLAY ROUND(67.12345, 5) FORMAT "99.99999". // 67.12345
DISPLAY ROUND(67.12345, 4) FORMAT "99.99999". // 67.12350
DISPLAY ROUND(67.12345, 3) FORMAT "99.99999". // 67.12300
DISPLAY ROUND(67.12345, 2) FORMAT "99.99999". // 67.12000
DISPLAY ROUND(67.12345, 1) FORMAT "99.99999". // 67.10000
DISPLAY ROUND(67.12345, 0) FORMAT "99.99999". // 67.00000

TRUNCATE Truncates a decimal expression to a specified number of decimal places, returning a decimal value.

TRUNCATE( number, places)

DISPLAY TRUNCATE(67.12345, 6) FORMAT "99.99999". // 67.12345
DISPLAY TRUNCATE(67.12345, 5) FORMAT "99.99999". // 67.12345
DISPLAY TRUNCATE(67.12345, 4) FORMAT "99.99999". // 67.12340
DISPLAY TRUNCATE(67.12345, 3) FORMAT "99.99999". // 67.12300
DISPLAY TRUNCATE(67.12345, 2) FORMAT "99.99999". // 67.12000
DISPLAY TRUNCATE(67.12345, 1) FORMAT "99.99999". // 67.10000
DISPLAY TRUNCATE(67.12345, 0) FORMAT "99.99999". // 67.00000

ABSOLUTE - Returns the absolute value of a number

DISPLAY ABS(10 - 12). //Displays 2
DISPLAY ABS(-2) = ABS(2). //Displays yes

MINIMUM and MAXIMUM - returns the smallest and largest number

MINIMUM(number1, number2, ... numbern)

MAXIMUM(number1, number2, ... numbern)

DEFINE VARIABLE i AS INTEGER     NO-UNDO.
DEFINE VARIABLE j AS INTEGER     NO-UNDO.
DEFINE VARIABLE k AS INTEGER     NO-UNDO.

i = 40.
j = 45.
k = 56.

DISPLAY MINIMUM(i, j, k) MAXIMUM(i, j, k). // Displays 40 and 56

Comparing numbers

https://riptutorial.com/
There are standard functions built in for comparing equality, inequality etc.

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
<th>Alternative</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal</td>
<td>=</td>
<td>EQ</td>
<td>i = j</td>
</tr>
<tr>
<td>Not equal</td>
<td>&lt;&gt;</td>
<td>NE</td>
<td>i &lt;&gt; j</td>
</tr>
<tr>
<td>Less than</td>
<td>&lt;</td>
<td>LT</td>
<td>i &lt; j</td>
</tr>
<tr>
<td>less than or equal</td>
<td>&lt;=</td>
<td>LE</td>
<td>i &lt;= j</td>
</tr>
<tr>
<td>Greater than</td>
<td>&gt;=</td>
<td>GT</td>
<td>i &gt; j</td>
</tr>
<tr>
<td>Greater than or equal</td>
<td>&gt;=</td>
<td>GE</td>
<td>i &gt;= j</td>
</tr>
</tbody>
</table>

The symbol can be exchanged with the alternative and vice versa. So \( \text{var1} <> \text{var2} \) is the same thing as \( \text{var1} \text{ NE} \text{ var2} \).

You can compare a float with an integer but you cannot compare for instance a date with an integer.

**Random number generator**

**RANDOM** - generates a random number

\[
\text{RANDOM}(\text{low}, \text{high})
\]

Generates a pseudo random integer between low and high

// Example that generates 20 random numbers between 1 and 20 (1 and 20 included)
DEFINE VARIABLE i AS INTEGER NO-UNDO.

DO i = 1 TO 20.
   DISPLAY i RANDOM(1, 20).
   PAUSE.
END.

Read Working with numbers online: https://riptutorial.com/progress-4gl/topic/8878/working-with-numbers
# Credits

<table>
<thead>
<tr>
<th>S. No</th>
<th>Chapters</th>
<th>Contributors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Getting started with progress-4gl</td>
<td>Community, Jensd, Stephen Leppik</td>
</tr>
<tr>
<td>2</td>
<td>Compiling</td>
<td>Jensd</td>
</tr>
<tr>
<td>3</td>
<td>Conditional statements</td>
<td>Jensd</td>
</tr>
<tr>
<td>4</td>
<td>FIND statement</td>
<td>Jensd</td>
</tr>
<tr>
<td>5</td>
<td>Functions</td>
<td>Jensd</td>
</tr>
<tr>
<td>6</td>
<td>Iterating</td>
<td>Jensd</td>
</tr>
<tr>
<td>7</td>
<td>OS-utilities</td>
<td>Jensd</td>
</tr>
<tr>
<td>8</td>
<td>Procedures</td>
<td>Jensd</td>
</tr>
<tr>
<td>9</td>
<td>Queries</td>
<td>Jensd, R3uK</td>
</tr>
<tr>
<td>10</td>
<td>Strings</td>
<td>Jensd</td>
</tr>
<tr>
<td>11</td>
<td>TEMP-TABLE</td>
<td>Jensd</td>
</tr>
<tr>
<td>12</td>
<td>Variables</td>
<td>Jensd</td>
</tr>
<tr>
<td>13</td>
<td>Working with numbers</td>
<td>Jensd</td>
</tr>
</tbody>
</table>