The Progress Electronic Magazine

In this issue:

Publisher's Statement:

In the last E-Zine, I stated "One of the things Peter doesn't include, is the name of the program that did the change – my little contribution of should be's for the article!" Turns out in his production system he DOES so, and a few other things that was not mentioned in the article. My oops!

Also, as I mentioned on the progress E-Mail list, I mentioned the beginning of a series of articles for natural language processing. To make it a bit more fun of a read, we will use it within an old time text adventure game. In this issue, we look at the basics of describing the world the player will be living in and the data structures to describe that. The next issue will look at the natural language interface used to manipulate those data structures.

Finally, John Green has brought along an article about using Prolint – a freely available open source application to aid in the development of Progress language based applications. Included also is a description and how to of his company's proparse tool in a product article.

To your success,
Scott Auge
Founder, Amduus Information Works, Inc.
sauge@amduus.com

Coding Article: Natural Language Interfacing and Adventure Game Part 1

Written by Scott Auge sauge@amduus.com

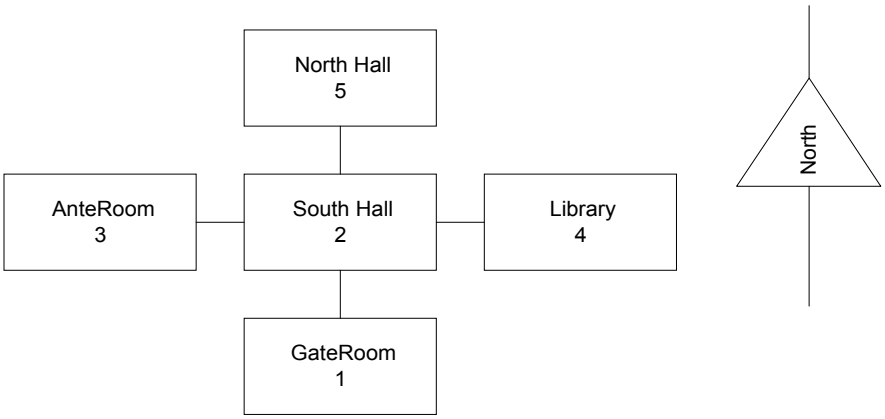
What is an adventure game?

For the young ones out here, there was a time, long time ago – 1980's long ago, when games were played without graphics. In these games, called adventure games, the player used English phrases to interact with a game representing a world. The game would allow the user to maneuver around a map of places, as well manipulate objects and interact with various creatures encountered in the world.

This is simple enough where we do not need to explain and develop a lot of technology and algorithms, but yet allow us to get to the point of using a Natural Language Interface to interact with a Progress based application.

Representing Rooms In The Game

Below is a simple map that we will use to allow the user to maneuver around.



Map of places in the game

Each place a player can reside is represented by a Room. Each Room has a name and a number representing it. We can use a graphing matrix to represent the places and connections between them as follows:

Room	*North*	*South*	*East*	*West*
1	2	0	0	0
2	5	1	4	3
3	0	0	2	0
4	0	0	0	2
5	0	2	0	0

One reads this as follows, I am currently in Room number n. If I want to go south, I look in the column South and that will put me in Room m. Very simple. As you can guess, the table for the Room Map is simply a definition of:

```
============================== Table: RoomMap ==============================

              Table Flags: "f" = frozen, "s" = a SQL table


Table                          Dump     Table Field Index Table
Name                           Name     Flags Count Index Label
------------------------------ -------- ----- ----- ----- --------------------
RoomMap                        roommap            5     1 ?

  Description: Map of rooms available in an adventure game
 Storage Area: Schema Area


============================== FIELD SUMMARY ==============================
============================== Table: RoomMap ==============================

Flags: <c>ase sensitive, <i>ndex component, <m>andatory, <v>iew component

Order Field Name                 Data Type   Flags Format          Initial
----- ------------------------- ----------- ----- --------------- ----------
   10 RoomNumber                inte        i     ->,>>>,>>9      0
   20 North                     inte              ->,>>>,>>9      0
   30 South                     inte              ->,>>>,>>9      0
   40 East                      inte              ->,>>>,>>9      0
   50 West                      inte              ->,>>>,>>9      0

Field Name                     Label                Column Label
------------------------------ -------------------- --------------------
RoomNumber                     RoomNumber           RoomNumber
North                          North                North
South                          South                South
East                           East                 East
West                           West                 West
```

```
============================== INDEX SUMMARY ==============================
============================== Table: RoomMap ==============================

Flags: <p>rimary, <u>nique, <w>ord, <a>bbreviated, <i>nactive, + asc, - desc

Flags Index Name                       Cnt Field Name
----- -------------------------------- --- --------------------------------
pu    pukey                              1 + RoomNumber

** Index Name: pukey
 Storage Area: Schema Area
```

Needless to say, to add more depth to the game, each room should have a description. Below is the table to hold the description of the room.

```
============================== Table: RoomDesc ==============================

              Table Flags: "f" = frozen, "s" = a SQL table


Table                          Dump     Table Field Index Table
Name                           Name     Flags Count Count Label
------------------------------ -------- ----- ----- ----- --------------------
RoomDesc                       roomdesc           2     1 ?

  Description: Description of rooms in an adventure game
 Storage Area: Schema Area


============================== FIELD SUMMARY ==============================
============================== Table: RoomDesc ==============================

Flags: <c>ase sensitive, <i>ndex component, <m>andatory, <v>iew component

Order Field Name                Data Type   Flags Format          Initial
----- ------------------------- ----------- ----- --------------- ----------
   10 RoomNumber                inte        i     ->,>>>,>>9      0
   20 Description               char              x(80)

Field Name                     Label                  Column Label
------------------------------ ---------------------- ----------------------
RoomNumber                     RoomNumber             RoomNumber
Description                    Description            Description


============================== INDEX SUMMARY ==============================
============================== Table: RoomDesc ==============================

Flags: <p>rimary, <u>nique, <w>ord, <a>bbreviated, <i>nactive, + asc, - desc

Flags Index Name                       Cnt Field Name
----- -------------------------------- --- --------------------------------
pu    pukey                              1 + RoomNumber
```

```
** Index Name: pukey
 Storage Area: Schema Area
```

I leave code to edit these tables up to the readership – perhaps someone will be willing to contribute!

Representing where the player is in the game

The player's position in the map is simply held in a variable – PlayerCurrentRoom. Within the variable is the room number the player is in. To move the player into a different room, re-assign the variable. Later on, you will realize that we actually don't keep this in a variable but in a database table. There is a reason to do this more so than because we are using a web interface.

Pulling it together

We now need an interface screen to basically determine where the user is at via the PlayerCurrentRoom variable, bring up the appropriate RoomDesc record and display it on the screen, as well receive input from the player. We do this in a simple web page called Game.html.

```
<!--WSS

DEF VAR cRoomDescription    AS CHARACTER NO-UNDO.
DEF VAR cRoomInventory      AS CHARACTER NO-UNDO.

DEF VAR cPlayerCurrentRoom  AS CHARACTER NO-UNDO.
DEF VAR iPlayerCurrentRoom  AS INTEGER NO-UNDO.

DEF VAR cCommand            AS CHARACTER NO-UNDO.

/* Input what the player wants to do */

ASSIGN cCommand = GET-VALUE("Command").

/* Determine where player was at */

ASSIGN cPlayerCurrentRoom = GET-VALUE("PlayerCurrentRoom").
ASSIGN iPlayerCurrentRoom = INT(cPlayerCurrentRoom).

/* Determine what the user's command is all about */

RUN game/ParseCommand.p
(INPUT cCommand).

/* Determine the current room.  If none (such as first time in the game  */
/* then default to room one.  Scotta as the sessionid is just hard coded */
/* for simplicities sake - this should be the user's login.              */

FIND WebState NO-LOCK
WHERE WebState.SessionID = "scotta"
  AND WebState.Category = "Game"
  AND WebState.Name = "PlayerCurrentRoom"
NO-ERROR.
```

```
IF AVAILABLE WebState THEN
  ASSIGN iPlayerCurrentRoom = INT(WebState.Data).
ELSE
  ASSIGN iPlayerCurrentRoom = 1.

/* Pull up the appropriate room description */

FIND RoomDesc NO-LOCK
WHERE RoomDesc.RoomNumber = iPlayerCurrentRoom
NO-ERROR.

IF AVAILABLE RoomDesc THEN
  ASSIGN cRoomDescription = RoomDesc.Description.

-->

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>Game</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>

<body bgcolor="#FFFFFF">

<p> </p>
<p align="center">Natural Language Processing</p>
<p align="center">Adventure Game</p>
<form name="form1" method="post" action="">
<input type="hidden" value="`cPlayerCurrentRoom`" name="PlayerCurrentRoom">
  <table width="90%" border="0" align="center" bgcolor="#CCCCCC">
    <tr>
      <td> </td>
    </tr>
    <tr>
      <td><table width="90%" border="0" align="center" bgcolor="#FFFFFF">
          <tr>
            <td><p align="center">`cRoomDescription`</p>
              <p align="center">`cRoomInventory`</p></td>
          </tr>
        </table></td>
    </tr>
    <tr>
      <td><center>
          <input name="Command" type="text" id="Command" size="90">
        </center></td>
    </tr>
    <tr>
      <td><center>
          <input type="submit" name="Submit" value="Submit">
        </center></td>
    </tr>
  </table>
</form>
<p> </p>
</body>
</html>
```

With Game.html we spit out the room description and a prompt for the user to tell us what they wish to do. Very simple. What is not simple is the underlying logic underneath the prompt. That is what we will discuss in the next E-Zine!

About the author: Scott Auge is the founder of Amduus Information Works, Inc. He has been programming in the Progress environment since 1994. His works have included E-Business initiatives and focuses on web applications on UNIX platforms.
sauge@amduus.com

Product Article: Good News, Bad News, Good News

By John Green, www.joanju.com, john@joanju.com

The good news is, you found out from your sales manager that you are going to sell a license of your software to that big foreign firm. The bad news is, you've just been told by the development manager to go through the entire system, and find all expressions similar to this:

```
DISPLAY
  "The sales rep for " + cust-name + " is " + salesrep + "."
  FORMAT "x(60)".
```

And change them so that they look like this:

```
DISPLAY
  SUBSTITUTE("The sales rep for &1 is &2.", cust-name, salesrep)
  FORMAT "x(60)".
```

You have to do this because the string in the second expression is easier to translate than the strings in the first expression. Let's see: There's 4500 source files in the application, this should only take you about three weeks or so. Bletch. There must be a better way.

Tedious source code changes like this should be automated or assisted by machine. For one thing, just finding all lines of code which need to be changed will require programmers to read through the entire application, because you can't use "grep" or an XREF database to find expressions like this.

For another thing, if you have to make dozens (or hundreds!) of changes like this, you are bound to make a few mistakes.

Finally, having programmers do this by hand is going to take a long time (i.e. it's expensive!), and the programmers aren't going to be particularly happy about spending three weeks doing this boring task.

We (www.joanju.com) have had recent success with automating this sort of transformation, using our product "Proparse".

Here's a new term in case you haven't seen it before: *refactoring* is the process of improving or restructuring source code without changing the behaviour of the application.

How do we do it? Well, the first step is to just find all lines of code which need to be changed. Prolint (www.global-shared.com/prolint/) has already been doing that for a while now. Prolint has more than 40 "rules" for finding issues in your source code such as this one, and the rule named "substitute" is the one that we are interested in here.

Like most rules in Prolint, the "substitute" rule uses the results from Proparse in order to find particular instances of source code. In this case, it finds all expressions which contain multiple "+" nodes as well as multiple translatable strings. If it finds an instance of that, it reports it.

That's easy enough, but what about automated refactoring? That turns out to be reasonably straightforward as well. Proparse generates a tree (an *abstract syntax tree*, or AST) and each node in the tree represents some token from the source code, with some extra nodes thrown in for tree organization. For this transformation, it's just a matter of adding, changing, and moving a few nodes around. Have another look at the expression before and after the transformation, and you might be able to imagine an algorithm for automating the change.

But what if it is a more complex expression, like:

```
"The total is " + STRING(subtotal * tax-rate) + " Euros."
```

That is why we work with a syntax tree, rather than just with a list of tokens. For an expression like `STRING(subtotal * tax-rate)`, the STRING node is the topmost node in a branch, and all of the other tokens in `(subtotal * tax-rate)` are children and grandchildren nodes of the STRING node. So, when we want to change things around to look like this:

```
SUBSTITUTE("The total is &1 Euros.", STRING(subtotal * tax-rate))
```

…we just have to grab the STRING node and move it around, dragging the entire branch with it. Easy!

Can we build you refactoring tools for your specific needs? You bet! Drop us an email, we might surprise you with what we're capable of.

> *John has been programming in Progress since 1989, and has been lead developer and team leader on various ERP systems, as well as on Roundtable at Starbase. Recently he has moved over to the C side (seaside?) in order to develop Proparse. John Green, www.joanju.com, john@joanju.com*

Coding Article: Programming Prolint by Example: "noeffect"

John Green: www.joanju.com, john@joanju.com
With contributions from Jurjen Dijkstra: jurjen@global-shared.com
Corrections and suggestions from Judy Hoffman Green: judy@joanju.com

Yes, Prolint is open source, and yes, Jurjen is very smart, but no, adding new rules to Prolint is **not** rocket science!

Now that Prolint has been around for a while, we are starting to see various categories of Prolint rules emerging. Some find potential performance problems, some find problems in the code which will make translation (i18n) more difficult, some find problems which would make it more difficult to use dataservers, and some find problems which might just downright be a **bug**.

We'll have a look here at a rule which falls into the bug-finding category: "noeffect". Try the following code:

```
def var x as int init 1.
x eq 12.
display x.
```

In case you are wondering: yes, this sort of bug actually has been found in production code.

The statement "x eq 12." is actually just an expression. It compares x to 12, evaluates to "false", and that's it. It doesn't do anything. It has no effect.

The parser makes it easy to find statements which are just expressions. Every statement which becomes a branch in the syntax tree generated by the parser must have a "head" node. For a statement like

```
DISPLAY "hello".
```

the DISPLAY node is the head node, and all other tokens in the display statement are child or grandchild nodes to the DISPLAY node.

For a statement which is just an expression, Proparse adds a “synthetic” node into the tree, just for tree organization. The synthetic node is an “Expr_statement” node, and it becomes the head node for the statement.

Finding all Expr_statement nodes gives us a starting point. Let’s start looking at the code, starting with some comments about the comments:

```
/* ------------------------------------------------------------------------
   file    :  prolint/rules/noeffect.p
   by      :  John Green
   purpose :  Find statements which are nothing but expressions.
              Of those, evaluate which /cannot/ have an effect.
              User-defined functions might have an effect,
              and methods might have an effect.
              Of the built-in functions, I am only aware of the following
              having any effect:
              DYNAMIC-FUNCTION
              ETIME
              SETUSERID
              SUPER
              CURRENT-LANGUAGE
    ------------------------------------------------------------------------

    Copyright (C) 2001,2002 John Green

    This file is part of Prolint.

    Prolint is free software; you can redistribute it and/or
    modify it under the terms of the GNU Lesser General Public
    License as published by the Free Software Foundation; either
    version 2.1 of the License, or (at your option) any later version.

    Prolint is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public
    License along with Prolint; if not, write to the Free Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
   ------------------------------------------------------------------------
*/
```

This standard text needs to be included in the header, it is a requirement of LPGL.
Prolint is open-source and GPL, under the “Lesser GPL”, which means that it can be included in commercial products, but you must include the source and you just can’t claim it to be your own. Or something like that. You should not think of this as a limitation of your rights, instead the LPGL provides the benefit that every improvement to Prolint will always be publicly available with source. (Proparse is not free though – due to that “pay the rent” thing.)

```
{prolint/ruleparams.i}
```

Here we included the basic configuration and setup required for each Prolint rule. We don't need to know much about what goes on in there to write a basic rule, except that it is the source of some of the variables which we will use below.

```
RUN searchNode {&insuper}
    (hTopnode,                /* "Program_root" node */
     "InspectNode":U,         /* name of callback procedure */
     "Expr_statement":U).     /* list of statements to search, ?=all */

RETURN.
```

Here we ran Prolint's search routine, which is found in the Prolint super procedure, `lintsuper.p`.

The first parameter it requires is the starting point in the syntax tree generated by Proparse. In almost all cases, we just want to start at the very "root" of the tree – the topmost node. The variable `hTopnode` is of course made available to us by `ruleparams.i` and it is a pointer to the topmost node in the syntax tree. That's probably even more than we really need to know.

The procedure `searchNode` uses a *callback* mechanism. So, `searchNode` does the search, and with the results of that search, it runs what we tell it to run. In our case, we want it to run `InspectNode`, which is an internal procedure in this .p, which we'll describe below. That's the second parameter.

How does `searchNode` know when it should call `InspectNode`? When it finds an `Expr_statement` node within the syntax tree. That's the final parameter.

That's it for getting Prolint to find us all of the nodes that we might be interested in. Now let's look at where the interesting work is done.

```
PROCEDURE InspectNode:
  /* purpose: see if the statement in theNode is
     really a statement without effect */
  DEFINE INPUT  PARAMETER theNode        AS INTEGER NO-UNDO.
  DEFINE OUTPUT PARAMETER AbortSearch    AS LOGICAL NO-UNDO INITIAL NO.
  DEFINE OUTPUT PARAMETER SearchChildren AS LOGICAL NO-UNDO INITIAL NO.
```

Every callback procedure for `searchNode` requires three parameters. The first is an integer handle for the node that it found. The integer handle was set up by `searchNode`, and that handle is meaningful to the parser. In our case here, `theNode` will be a handle to an Expr_statement node.

For this rule, we don't use `AbortSearch` or `SearchChildren`. We'll talk about those parameters in future articles about Prolint architecture and other Prolint rules.

Now let's get back to one of our goals: we want to find all Expr_statement nodes, but there are certain expression statements which really **do** have an effect, and we don't want Prolint to raise warnings about those.

```
DEFINE VARIABLE qname       AS CHARACTER NO-UNDO INITIAL "rule_noeffect":U.
DEFINE VARIABLE i           AS INTEGER NO-UNDO.
DEFINE VARIABLE numResults  AS INTEGER NO-UNDO.
DEFINE VARIABLE result      AS INTEGER NO-UNDO.

result = parserGetHandle().

check-block:
DO:

  IF parserQueryCreate(theNode, qname, "USER_FUNC":U) > 0 THEN
     LEAVE check-block.
  IF parserQueryCreate(theNode, qname, "DYNAMICFUNCTION":U) > 0 THEN
     LEAVE check-block.
  IF parserQueryCreate(theNode, qname, "ETIME":U) > 0 THEN
     LEAVE check-block.
  IF parserQueryCreate(theNode, qname, "SETUSERID":U) > 0 THEN
     LEAVE check-block.
  IF parserQueryCreate(theNode, qname, "SUPER":U) > 0 THEN
     LEAVE check-block.
```

(Here we have run into direct calls to some of Proparse's "parser" functions. If you would like to know more beyond the fact that Proparse function names all begin with "parser", please see www.joanju.com/whitepapers/, and look for the link to "Proparse API Quick Introduction".)

We query for specific types of nodes within the Expr_statement branch. Each time the query is built, it returns the number of results, and if there are any results, then we skip out of `check-block` without issuing a warning. As described in the comments at the top of the program, user defined functions, as well as some built-in functions, are actually functions that perform some action. Most built-in functions only evaluate something and return a value - they do not actually change the state of the application.

For a better understanding of the next code snippet, let's first have a look at two example statements:

```
/* hEditor is a handle to an editor widget.
   We want to switch the editor to read-only: */
hEditor:READ-ONLY.

/* hQuery is a handle to a dynamic query.
   We want it to fetch the first record: */
hQuery:GET-FIRST().
```

The first statement has no effect, because READ-ONLY is not a method, it is only a property. We want Prolint to raise a "noeffect" warning for this statement because the programmer probably

made a mistake. The second statement actually does have effect (because it fetches a record into the buffer) so we do not want to raise a warning.
So how can we tell if something is a property and not a method? To answer this, it is necessary to use the Proparse Tokenlister tool and see how Proparse translates these statement into tokens:

```
Expr_statement
    Widget_ref
        Field_ref
            ID    hEditor
        OBJCOLON    :
        READONLY    READ-ONLY
    PERIOD    .

Expr_statement
    Widget_ref
        Field_ref
            ID    hQuery
        OBJCOLON    :
        ID    GET-FIRST
        Method_param_list
            LEFTPAREN    (
            RIGHTPAREN    )
    PERIOD    .
```

The Tokenlister shows us that we can recognize a property because it contains an “OBJCOLON” token while it does not contain a “Method_param_list” token. Having learned this, let’s continue with the Prolint rule:

```
/* The last thing we check on is methods.
   a statement like  handle:READ-ONLY has no effect,
   a statement like  handle:GET-FIRST() does have effect,
   so look for OBJCOLON _not_ followed by node type "Method_param_list" */
ASSIGN
  result = parserGetHandle()
  numResults = parserQueryCreate(theNode, qname, "OBJCOLON":U).
DO i = 1 TO numResults:
  parserQueryGetResult(qname, i, result).
  /* from "widattr" in the tree spec:
   * (OBJCOLON . #(Array_subscript...)? #(Method_param_list...)? )+
   * First, move to the method or attribute name node - the .
   * (i.e. any) token after the OBJCOLON.
   * Then, simply  check next sibling twice.
   * First time might be Array_subscript.
   */
  parserNodeNextSibling(result, result).
  IF parserNodeNextSibling(result, result) = "Method_param_list":U THEN
     LEAVE check-block.
  IF parserNodeNextSibling(result, result) = "Method_param_list":U THEN
     LEAVE check-block.
END.
```

In the check made above, we are assuming that attributes do not perform actions, and methods do. For the most part, this assumption works quite well, and it is certainly close enough for the purpose of issuing warnings.

```
    /* If we got here, then the expression statement probably has no effect.
     * "Expr_statement" is a synthetic node, so it won't have a line
     * number. Instead, use the first non-synthetic node for PublishResult.
     */
    parserNodeFirstChild(theNode, result).
    DO WHILE parserGetNodeLine(result) EQ 0 :
       parserNodeFirstChild(result,result).
    END.
```

We've played a little trick here to find a node with a valid filename and line number. Some nodes in the parser's tree are inserted for tree structure only, and don't represent text from any source file. That is the case with an Expr_statement node. We just look for the first descendant node which does represent text from an actual sourcefile.

```
    RUN PublishResult {&insuper} (compilationunit,
                                  parserGetNodeFilename(result),
                                  parserGetNodeLine(result),
                                  "Statement has no effect":T,
                                  rule_id).

  END.  /* check-block */
```

Here we call Prolint's `PublishResult` procedure. `PublishResult` is responsible for sending a message to the active *output handler*. Prolint has various output handlers, and each of those is responsible for reporting a violation found in the code. Some output handlers report out to a text file, some report to the UI, some are geared towards putting the error messages into your favourite editor, and yet another reports the errors to a database for later analysis.

The arguments expected by the `PublishResult` procedure are:

- The name of the current compile unit (the program being linted)
- The name of the source file where the violation was found (may be an include file)
- The line number within the source file where the violation was
- The error message to display
- This rule's identifier – in this case, it is "noeffect"

The variables `compileationunit` and `rule_id` are defined and initialized via `ruleparams.i`.

```
  parserReleaseHandle(result).
  parserQueryClear(qname).

END PROCEDURE. /* InspectNode */
```

That's it! We tell the parser to release a couple of resources that we've grabbed, and we're done.

This rule had a few complications in it, and it actually requires more code than some of the simpler Prolint rules. However, I hope that it was still a good example for demonstrating the basic steps that are done in most Prolint rules:

- Include ruleparams.i
- Call searchNode with a node type to find and the name of a callback procedure
- Define the callback procedure
- The callback procedure may or may not perform further checks on the node that was found
- For nodes that are found to have actually violated some rule, call the PublishResult procedure

John Green: www.joanju.com, john@joanju.com
With contributions from Jurjen Dijkstra: jurjen@global-shared.com
Corrections and suggestions from Judy Hoffman Green: judy@joanju.com

John has been programming in Progress since 1989, and has been lead developer and team leader on various ERP systems, as well as on Roundtable at Starbase. Recently he has moved over to the C side (seaside?) in order to develop Proparse.

Letters:

Regarding the Auditing Article of Issue 24:

We implemented a very similar audit trail system a few years ago using the raw-transfer method to a raw field. It all worked very well until we upgraded our Progress installation from 8.2 to 9.1. Then we found that the RAW data transferred with 8.2 could not be transferred out using 9.1. It worked sometimes and then would not and was very unstable. The answers from Progress was that RAW-TRANSFER was not intended and a long term data storage format but only meant as a means of transferring data from one place to another.

We subsequently canned the system and used a BUFFER-COMPARE method to store only the fields that have changed in a CHAR field.

Robin Smith

Publishing Information:

Scott Auge publishes this document. I can be reached at sauge@amduus.com.

Amduus Information Works, Inc. assists in the publication of this document by providing an internet connection and web site for redistribution:

Amduus Information Works, Inc.
1818 Briarwood
Flint, MI 48507
http://www.amduus.com

Other Progress Publications Available:

This document focuses on the programming of Progress applications. If you wish to read more business oriented articles about Progress, be sure to see the Profile's magazine put out by Progress software:

http://www.progress.com/profiles/

There are other documents/links available at www.peg.com.

Products Available From Amduus:

Amduus Information Works, Inc. is a Progress reseller and ASPen partner. We primarily develop UNIX/Linux based applications for the web. We also perform integration of Progress applications through such languages and tools as MQ Series, C, and C++.

Amduus provides support for the following applications: Blue Diamond, Denkh, Denkh HTML Reporter, Red Arrow Portal (CMS), Survey Express and other software.

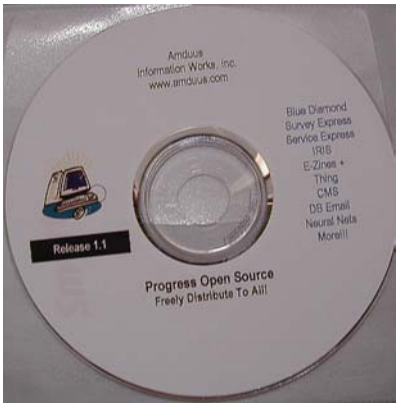
Article Submission Information:

Please submit your article in Microsoft Word format or as text. Please include a little bit about yourself for the About the Author paragraph.

Looking for technical articles, *marketing Progress* articles, articles about books relevant to programming/software industry, white papers, etc.

Order Form for Progress Open Source CD-ROM

COUPON 001A



This is an offer for the CD-ROM at lower than list savings!
This is a great way to support the E-Zine too!

Mail this form to:
Amduus Information Works, Inc.
1818 Briarwood
Flint, MI 48507

Please send _____ copies of the Open Source CD-ROM at $25.00 per disk to:

Name _____
Company _____
Address _____
City _____
State _____ Country _____
Zip _____

Please make your checks/money orders out to: Amduus Information Works, Inc. Cash works too!
This offer only valid in the United States of America.

The CD-ROM includes (all source code included):

- Blue Diamond/IRIS – Webspeed alternatives
- Survey Express – easily create text templates of surveys and then have the program generate the web pages automatically
- Service Express – Web based Help Desk.
- The Progress E-Zines, books on learning to program in Webspeed (PDF/Word/HTML)
- Denkh HTML Reporter – web based report writer
- CMS – a web content management system
- DB Email – Use pop3 to download emails into a Progress database
- Neural Networks – experiments in spam recognition and text message classification
- Denkh – create PDF file reports for Webspeed/UNIX CHUI!
- More!