

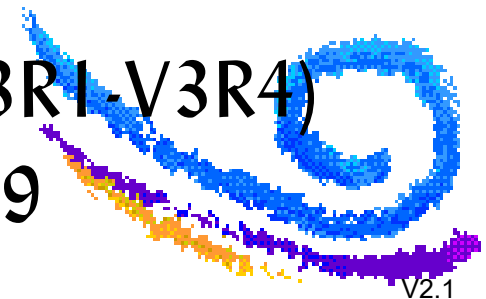
# **Enterprise COBOL: A Tool for Growth**

**COBOL Changes - Big Picture**

# Recent COBOL Compiler Versions

---

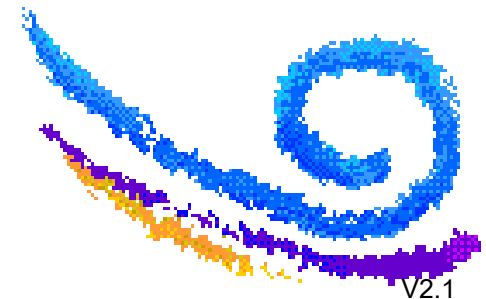
- ▶ OS/VS COBOL - (a.k.a. " COBOL I") no longer supported
- ▶ VS COBOL II - no longer supported (March 2001)
- ▶ All compilers after this require Language Environment (LE):
  - COBOL/370 - no longer supported (V1R1) (September 1997)
  - COBOL for MVS & VM - no longer supported (V1R2) (Dec. '01)
  - COBOL for OS/390 & VM (V2R1, V2R2) (Dec. '04)
  - **IBM Enterprise COBOL** for z/OS & OS/390 (V3R1-V3R4)
    - V4R1 December, 2007; V4R2 October, 2009



# Major New Features - General

---

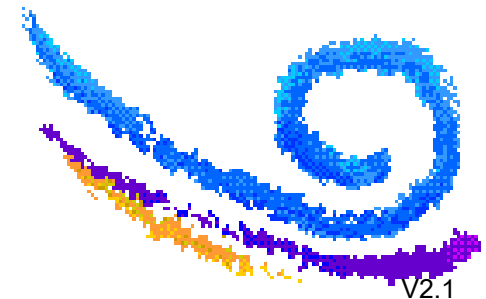
- ▶ Pointers and Address-of special register
- ▶ Reference modification (sub-stringing)
- ▶ Scope-terminators (END-*verb*)
- ▶ In-line PERFORM
- ▶ INITIALIZE statement
- ▶ EVALUATE statement
- ▶ Intrinsic function support
- ▶ Increased element and table size (**128MB**)



# Major New Features - General, continued

---

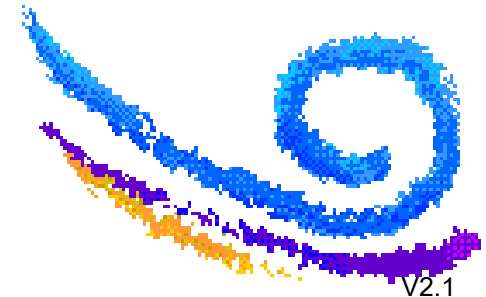
- ▶ Local-storage section
- ▶ Recursive programs
- ▶ DLL (Dynamic Link Library) support
  - Code DLLs in COBOL, call from other languages
  - Call DLLs from COBOL program
- ▶ Dynamic file allocation
- ▶ Support for HFS (Hierarchical File System) files
- ▶ Compile and run under z/OS UNIX
- ▶ DB2 co-processor
- ▶ **CICS co-translator**



# Major New Features - LE functions

---

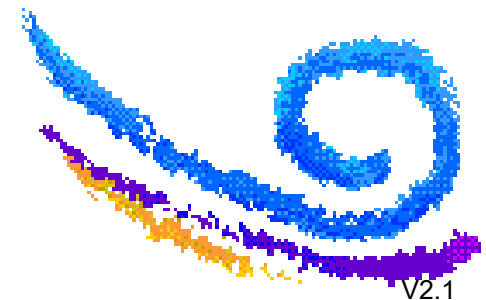
- ▶ Dynamic allocation of storage
  - Build large tables outside of load module
- ▶ Error handling
  - No need for Assembler STAE or SPIE routines
- ▶ New dump format
  - Ability to take snapshot memory dumps
- ▶ International services
  - Date, time, currency formatting, for example



# Major New Features - Improved interoperability with C

---

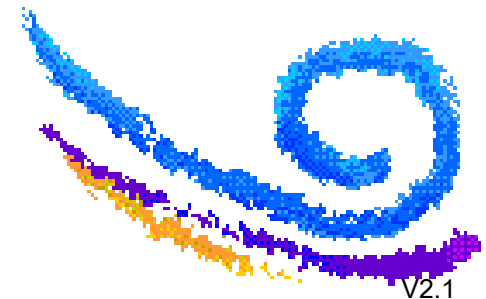
- ▶ Null-terminated strings
- ▶ Long names, mixed-case names
- ▶ Pass arguments in alternative ways
  - By reference - pointer to item
  - By content - pointer to copy of item
  - By value - copy of item in argument list directly
- ▶ Function-like behavior ("RETURNING" clause)



# Major New Features - Internet / Web oriented

---

- ▶ Unicode support - translate between codepages
- ▶ XML support
  - XML PARSE
    - XML PARSE VALIDATING WITH schema (V4R2)
  - XML GENERATE
- ▶ Object Oriented COBOL
  - Uses JVM (Java Virtual Machine)
- ▶ COBOL - Java Interoperability

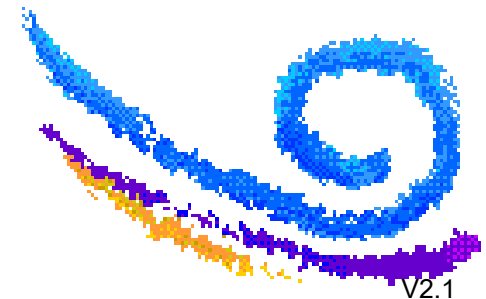


# COBOL Changes - format

---

- ▶ COBOL programs may be written in mixed case, since at least 1988
  - Just a style issue, but studies have shown this is easier to read:

```
Identification division.  
program-id.  exer01.  
*  Copyright (C) 2008 by Steven H. Comstock  
  
environment division.  
input-output section.  
file-control.  
    select zinputa assign to zinputa.  
    select rept assign to rept.  
  
data division.  
file section.  
fd  zinputa  
    block contains 0 records.  
01  zinputa-record          pic x(100).  
  
fd  rept.  
01  rept-rec                pic x(106).
```

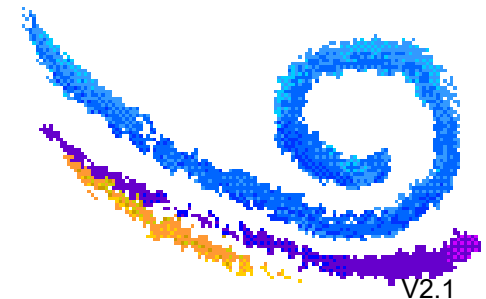


# COBOL Changes - format, continued

---

- ▶ The word "filler" is optional
  - Makes data structure stand out more cleanly:

```
working-storage section.  
  01  in-record.  
      05  in-part-number          pic x(9).  
      05  in-description          pic x(30).  
      05                               pic x(5).  
      05  in-unit-price           pic 9999v999.  
      05  in-quantity-on-hand    pic 99999.  
      05  in-quantity-on-ord     pic 999.  
      05  in-reorder-level       pic 999.  
      05  in-switch              pic xx.  
      05  in-old-part-no         pic x(9).  
      05  in-category            pic x(10).  
      05                               pic x(17).  
  
  01  rept-record.  
      05                               pic x(1) value spaces.  
      05  rept-part-number        pic x(9).  
      05                               pic x(3) value spaces.  
      05  rept-description        pic x(30).  
      05                               pic x(3) value spaces.  
      05  rept-quantity-on-hand  pic 99999.  
      05                               pic x(55) value spaces.
```

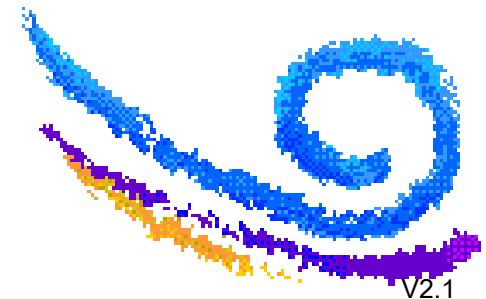


# COBOL Changes - format, continued

---

- ▶ Data types can now be specified as "packed-decimal" instead of "comp-3" and "binary" instead of "comp"
  - Again, a style issue, but it is more accurate / concise:

```
01  calc-stuff.  
02  number-items  pic s9(9)   binary          value +0.  
02  gross-sales   pic s9(v)99 packed-decimal value 0.00.  
02  tax-rate      pic s9v999  packed-decimal value 0.062.
```



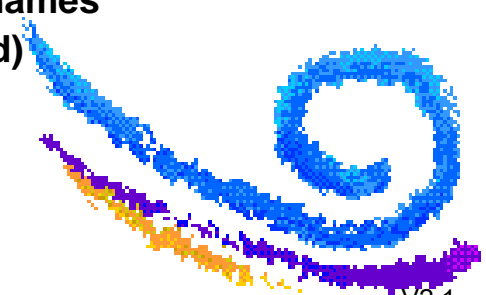
# COBOL Changes - format, continued

---

- ▶ You can use the underscore ( `_` ) instead of a dash ( `-` ) in user defined words (data items, paragraphs, sections, and so on)
  - In Enterprise COBOL V4R2 and later; *e.g.*:

```
01  calc_stuff.  
    02  number_items  pic s9(9)    binary          value +0.  
    02  gross_sales   pic s9(v)99  packed-decimal value 0.00.  
    02  tax_rate      pic s9v999   packed-decimal value 0.062.
```

This is also useful for processing XML from COBOL, since most XML names tend to have underscores instead of dashes (although both are allowed)

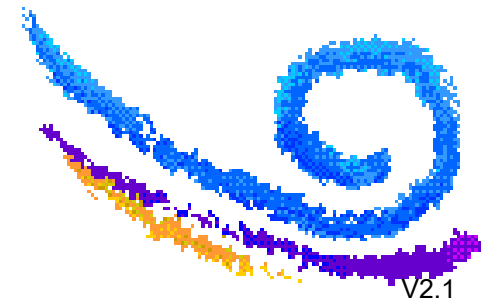


# COBOL Changes - Tables

---

- ▶ Maximum of 7 dimensions (up from 3)
- ▶ Maximum of **128MB** per table (up from 128K)
- ▶ Initialize table with VALUE at elementary level:

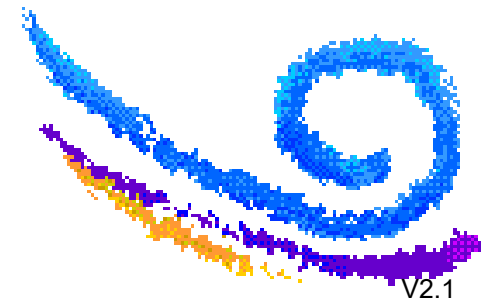
```
01 Sales-table.  
   05 Department occurs 20 times.  
       10 Dept-name pic x(20) value spaces  
       10 Dept-sales-amt pic s9(5)v99 packed-decimal value 0.
```



# COBOL Changes - Working-Storage

---

- ▶ The size of working storage has been expanded to **128MB** (up from 1MB)
  - Allows for more data-driven code and less fragmentation of routines, perhaps



# All Data Items Must Be Described

---

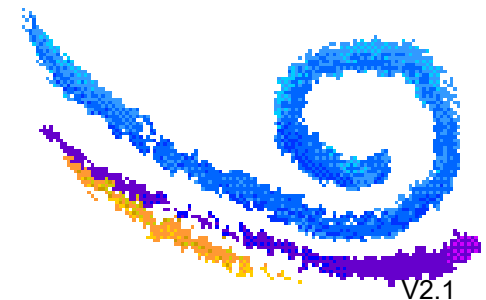
▶ Two clauses are relevant to the description:

- USAGE IS clause

- Strangely enough, both the words USAGE and IS are optional - the compiler looks for the actual allowed data types

- PICTURE clause

- Required for some usages; for others, it is not allowed

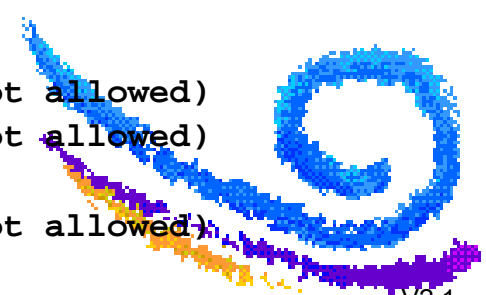


# USAGE Types

---

## ► Here is the complete list of reserved words for USAGE:

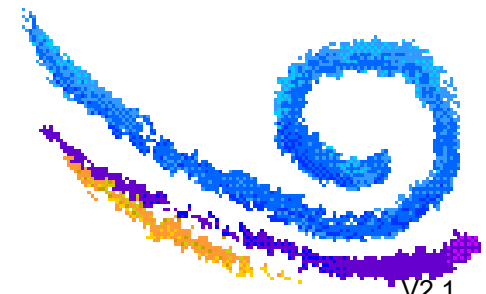
BINARY	- same as COMP and COMP-4; binary integer
COMP	- binary integer data
COMP-1	- single precision floating point (PIC not allowed)
COMP-2	- double precision floating point (PIC not allowed)
COMP-3	- packed decimal data
COMP-4	- binary integer data
NOTE: binary, comp, comp-4 are all the same; values in these kinds of fields may be truncated, based on compiler options and the particular work being done	
COMP-5	- binary integer, but will never be truncated
PACKED-DECIMAL	- same as comp-3
INDEX	- binary value: displacement into a table (PIC not allowed)
DISPLAY	- character string; single byte characters
DISPLAY-1	- character string; double byte characters
NATIONAL	- character string; UTF-16 (Unicode)
POINTER	- address of data item (PIC not allowed)
PROCEDURE-POINTER	- address of program entry point (8 bytes) (PIC not allowed)
FUNCTION-POINTER	- address of program entry point (4 bytes) (PIC not allowed)
OBJECT-REFERENCE	- handle for accessing a class in OO COBOL (PIC not allowed)



# PICTURE Clause

---

- ▶ A PICTURE (or simply PIC) clause is required for these data types:
  - Numeric data
    - Binary, comp, comp-4, comp-5
    - Comp-3, packed-decimal
  - String data
    - Display
    - Display-1
    - **National**



# PICTURE Clause, continued

---

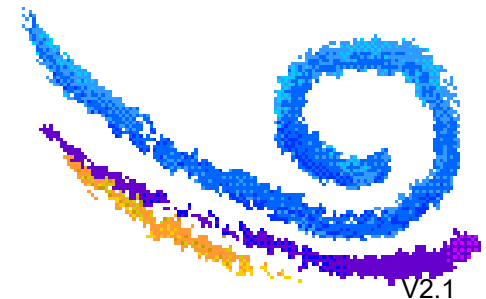
- ▶ A data item may have a PIC clause with no USAGE
  - In that case, the implied USAGE is
    - DISPLAY if the PIC has no G nor N items
    - DISPLAY-1 if the PIC has G items, or N items if the compiler option NSYMBOL is DBCS
    - NATIONAL if the PIC has N items and the compiler option NSYMBOL is NATIONAL (the default)

# Non-numeric Literals

---

- ▶ Non-numeric literals may be bounded by single quotes (apostrophes) or double quotes - even in the same program
  - As long as the opening delimiter is the same as the closing delimiter
  
- ▶ Independent of the QUOTE / APOST compiler setting
  - Which now just indicates what character to use for the figurative constant QUOTE[S]

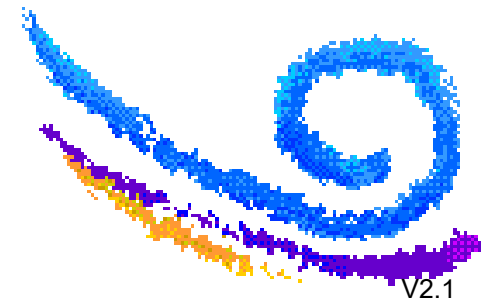
```
01 column-headers.  
02 pic x(10) value 'Employee Number'.  
02 pic x(3) value spaces.  
02 pic x(20) value "Last Name".
```



# Non-numeric Literal data - new options

---

- ▶ Hexadecimal literals: x'4A'
- ▶ Null terminated literals: z'Title of book'
- ▶ National literals: N'Heavy' (UTF-16)
- ▶ National hexadecimal literals: nx'672C'



# Reference modification

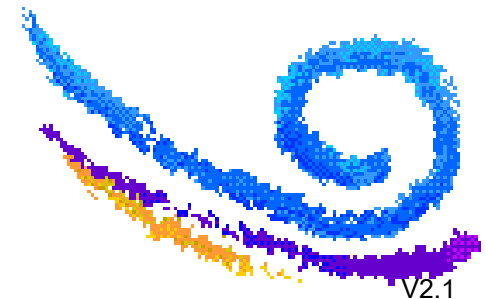
---

▶ *data-name(start:[length])* - examples:

```
move fielda to fieldb (5:7)
```

```
move fielda (N:3) to fieldb
```

```
move fielda (M+2): to fieldb (3: (xy-3)/2)
```



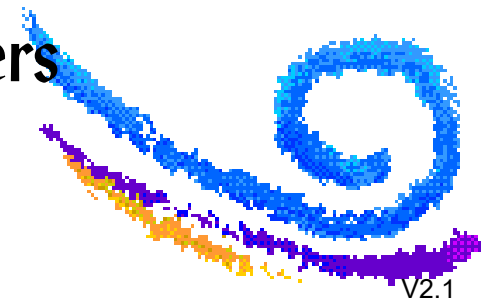
# Pointers and addresses

---

- ▶ A data item defined as pointer has an implicit definition of a 4 byte item intended to hold the address of some location in memory:

```
01 list-anchor      pointer.  
01 current-element pointer.  
01 no-more         pointer value null.
```

- ▶ Pointer data items may only be used in:
  - SET statements
  - Comparisons (and only for EQUAL or NOT EQUAL)
  - CALL - you may pass pointers as parameters

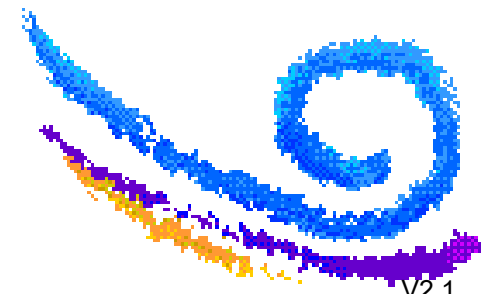


# Pointers and addresses, 2

---

- ▶ A pointer data item may only have a value of NULL
  - A reserved word meaning the value is not currently meaningful / valid
- ▶ Examples of using pointers:

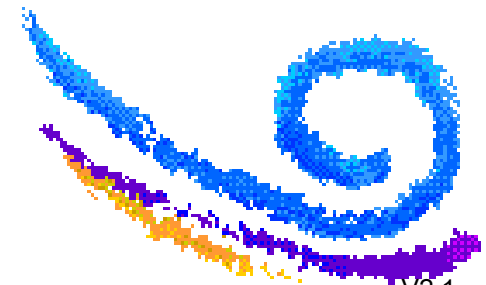
```
set current-element to list-anchor  
  
if list-anchor not = current-element ...  
  
set list-anchor to null  
  
call 'walker' using current-element
```



# Pointers and addresses, 3

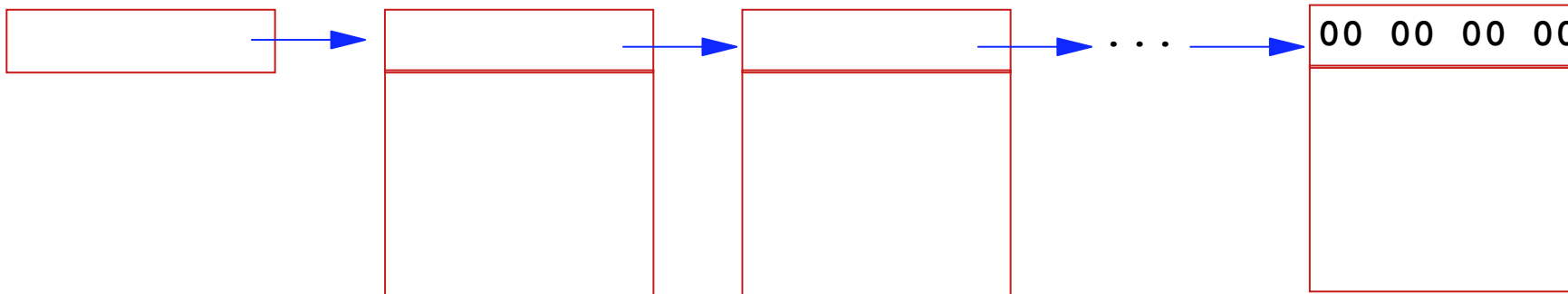
---

- ▶ You can reference the address of any data item
  - Level 01 and 77 items in linkage section can be sending or receiving fields
  - For all other data items, the address of the item can only be used for sending (locating) the item
- ▶ The Address of an item is a special register that is implicitly defined



# Pointers and addresses, 4

- ▶ Addresses and pointers can work together to process all the items in a linked list:



linkage section.

```
01 list-anchor pointer.
```

```
01 acct-rec.
```

```
02 next-rec pointer.
```

```
02 account-bal
```

```
pic s9(5)v99
```

```
packed-decimal.
```

procedure division using list-anchor.

```
set address of acct-rec to list-anchor
```

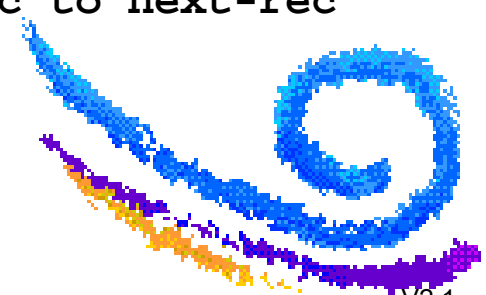
```
perform until next-rec is null
```

```
add account-bal to total-bal
```

```
set address of acct-rec to next-rec
```

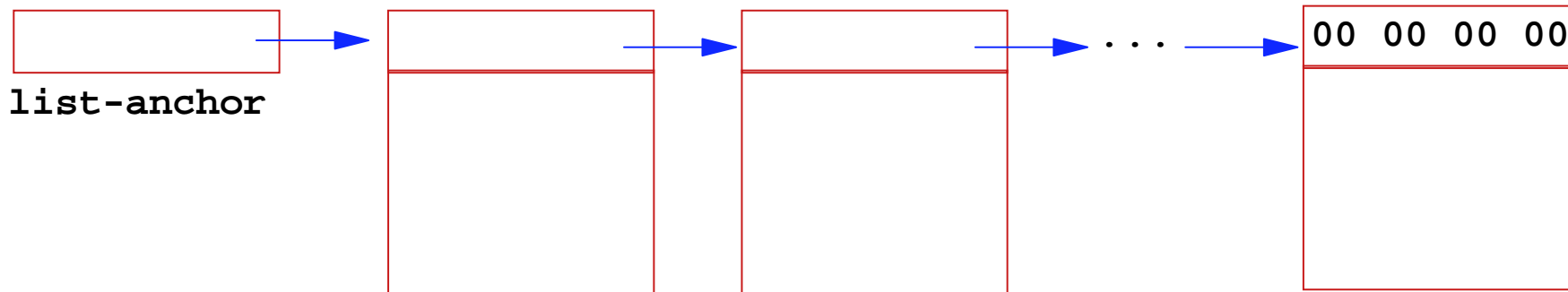
```
end-perform
```

Note also above: in-line perform



# Pointers and addresses, 5

- ▶ Addresses and pointers can work together to process all the items in a linked list:



linkage section.

```
01 list-anchor pointer.
```

```
01 acct-rec.
```

```
02 next-rec pointer.
```

```
02 account-bal
```

```
pic s9(5)v99
```

```
packed-decimal.
```

procedure division **using list-anchor**.

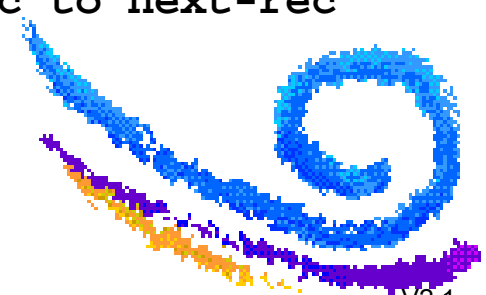
```
set address of acct-rec to list-anchor
```

```
perform until next-rec is null
```

```
add account-bal to total-bal
```

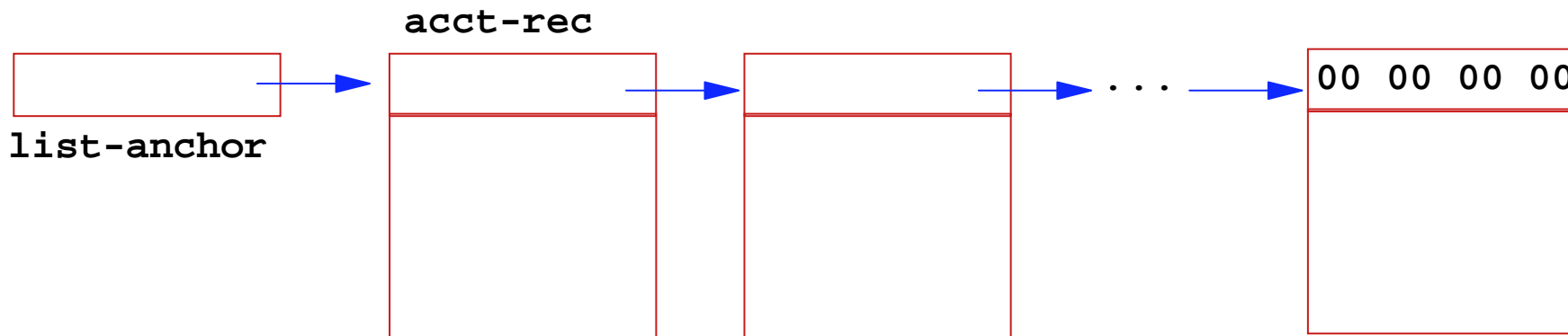
```
set address of acct-rec to next-rec
```

```
end-perform
```



# Pointers and addresses, 6

- ▶ Addresses and pointers can work together to process all the items in a linked list:



linkage section.

01 list-anchor pointer.

01 acct-rec.

02 next-rec pointer.

02 account-bal

pic s9(5)v99

packed-decimal.

procedure division using list-anchor.

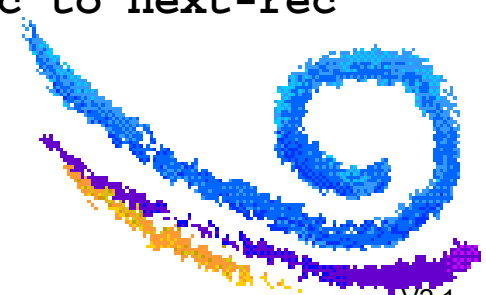
set address of acct-rec to list-anchor

perform until next-rec is null

add account-bal to total-bal

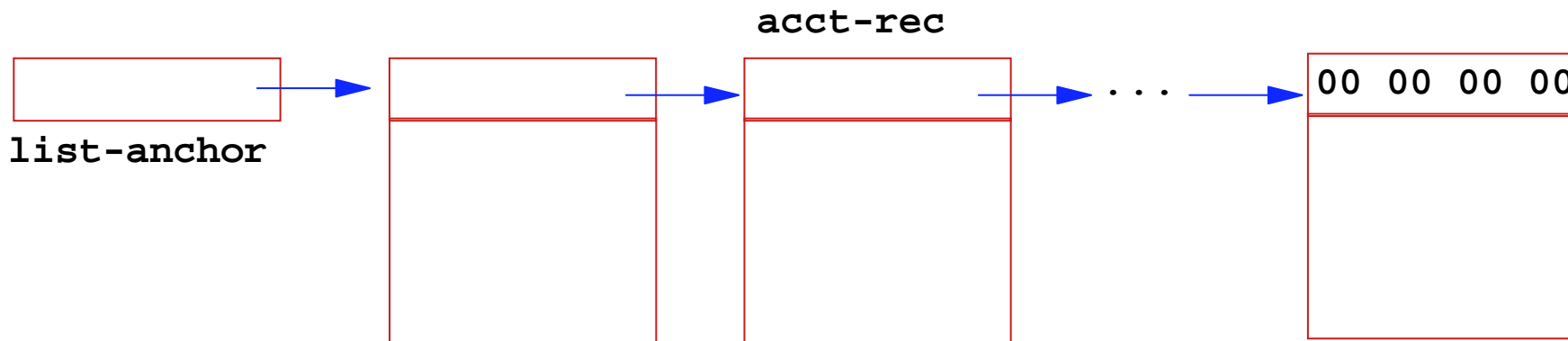
set address of acct-rec to next-rec

end-perform



# Pointers and addresses, 7

- ▶ Addresses and pointers can work together to process all the items in a linked list:



linkage section.

```
01 list-anchor pointer.
```

```
01 acct-rec.
```

```
02 next-rec pointer.
```

```
02 account-bal  
pic s9(5)v99  
packed-decimal.
```

procedure division using list-anchor.

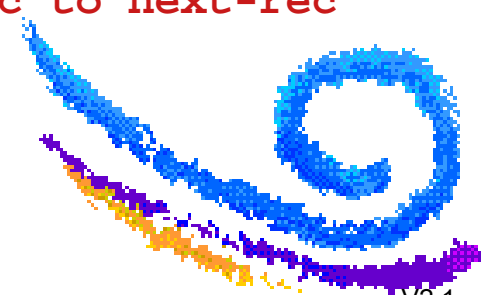
```
set address of acct-rec to list-anchor
```

```
perform until next-rec is null
```

```
add account-bal to total-bal
```

```
set address of acct-rec to next-rec
```

```
end-perform
```



# Scope terminators

## ► COBOL II introduced scope terminators to the language:

IF		...		END-IF
CALL	... ON OVERFLOW		...	END-CALL
CALL	... ON EXCEPTION	... NOT ON EXCEPTION	...	END-CALL
READ	... AT END	... NOT AT END	...	END-READ
READ	... INVALID KEY	... NOT INVALID KEY	...	END-READ
DELETE	... INVALID KEY	... NOT INVALID KEY	...	END-DELETE
REWRITE	... INVALID KEY	... NOT INVALID KEY	...	END-REWRITE
START	... INVALID KEY	... NOT INVALID KEY	...	END-START
WRITE	... END OF PAGE	... NOT END OF PAGE	...	END-WRITE
ADD	... ON SIZE ERROR	... NOT ON SIZE ERROR	...	END-ADD
COMPUTE	... ON SIZE ERROR	... NOT ON SIZE ERROR	...	END-COMPUTE
DIVIDE	... ON SIZE ERROR	... NOT ON SIZE ERROR	...	END-DIVIDE
MULTIPLY	... ON SIZE ERROR	... NOT ON SIZE ERROR	...	END-MULTIPLY
SUBTRACT	... ON SIZE ERROR	... NOT ON SIZE ERROR	...	END-SUBTRACT
RETURN	... AT END	... NOT AT END	...	END-RETURN
SEARCH	... AT END		...	END-SEARCH
STRING	... ON OVERFLOW	... NOT ON OVERFLOW	...	END-STRING
UNSTRING	... ON OVERFLOW	... NOT ON OVERFLOW	...	END-UNSTRING

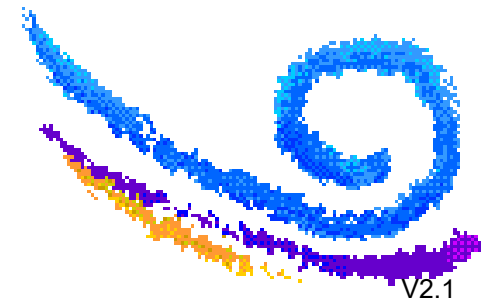
# Evaluate

---

- ▶ Along with end-if, the other statement programmers always pick up on, even if they had no training, is evaluate - it's just so useful

```
evaluate trans-id
  when 'A' perform add-trans
  when 'D' perform delete-trans
  when 'R'
  when 'U' perform update-trans
  when other perform bad-trans
end-evaluate
```

```
Evaluate trans-id
  when '1' perform trans-1-routine
  when '2' thru '4' perform trans-234-routine
  when 'A' perform trans-skip
  when not 'B' thru 'Z'
    perform other-numeric-trans
  when other perform trans-error
end-evaluate
```



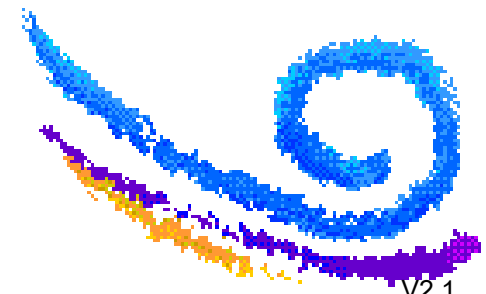
# Evaluate True

---

- ▶ But many don't realize you can evaluate True (or False) in addition to evaluating a data item:

```
Evaluate true
  when tax-collected = tax-owed
    perform audit-this-miracle
  when tax-collected < tax-owed
    perform initiate-collect
  when tax-collected > tax-owed
    perform initiate-refund
end-evaluate

Evaluate true
  when hourly-rate < min-wage
    compute hourly-rate = hourly-rate * 1.10
    perform min-message
  when hourly-rate > president-salary
    if president-salary > threshold-of-pain
      perform depose-current-president
      perform install-new-president
    end-if
  when other perform normal-salary-action
end-evaluate
```

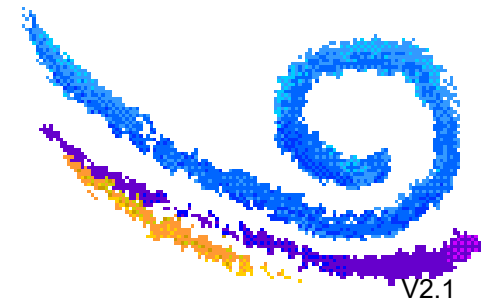


# Evaluate on conditional expressions

---

- ▶ You can also evaluate on complex expressions:

```
Evaluate one-a < one-b and hold-time < curr-time  
  when true perform gotcha  
  when false perform gotme  
end-evaluate
```



# Evaluate on condition names

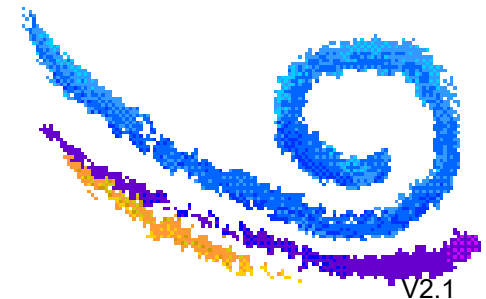
---

- ▶ You can also evaluate on level-88 names:

```
evaluate true
  when exempt continue
  when management perform human-sacrifice
  when hourly perform substantial-raise
  when other perform modest-raise.
```

\* This assumes something like:

```
01 Employee-type Pic X.
  88 exempt value 'E' 'P' 'Q'.
  88 management value 'M' 'V'.
  88 hourly value 'H' thru 'L'.
```



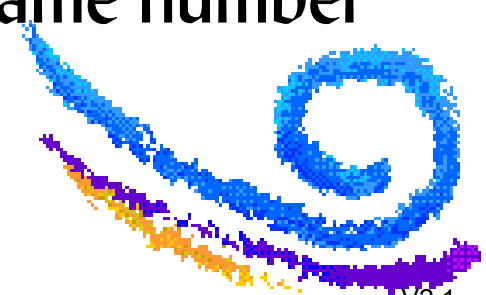
# Evaluate with Also

---

- ▶ "Also" is used to separate multiple tests

```
Evaluate co-type also co-size also co-color
  when '1' also 'BIG' also 'BLUE'
    perform found-IBM
  when '1' also 'TALL' also 'GREEN'
    perform found-Celtics
  when 'F' also 'HEAVY' also 'BLACK+BLUE'
    perform found-NY-Giants
  when 'Y' also 'LEAN' also 'PAISLEY'
    perform found-yuppie-co
  when other
    perform no-match-on-co
end-evaluate
```

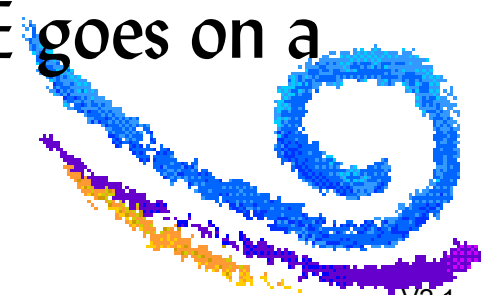
- Notice your WHEN clauses have to have the same number of ALSOs as your EVALUATE clause



# Continue

---

- ▶ With the advent of all the scope terminators, many programmers adopted a style of "minimum punctuation"
  - Which in the Procedure Division means: only periods needed are at the end of a paragraph or section header and the end of a paragraph or section body
  - Of course, this lead to a rash of errors, especially for code that used NEXT SENTENCE: NEXT SENTENCE goes on a hunt for the next period ...

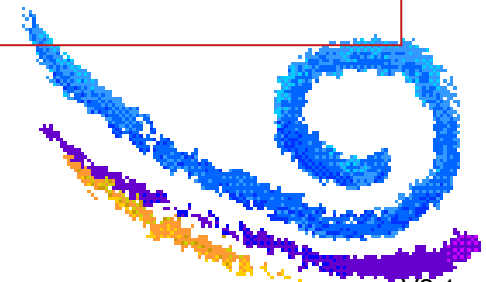


# Continue, 2

- ▶ When using the new style, the second example is almost certainly what you want:

```
Move 'ANTELOPE' to animal-type
If item = ideal-value
  move 'SAN DIEGO' to zoo-id
  if unit-cost > maximum-cost
    next sentence
  else
    move unit-cost to maximum-cost
  End-if
  Move 'AARDVARK' to animal-type
else
  perform item-check
End-if
Move animal-type to print-animal-type.
Stop run.
```

```
Move 'ANTELOPE' to animal-type
If item = ideal-value
  move 'SAN DIEGO' to zoo-id
  if unit-cost > maximum-cost
    continue
  else
    move unit-cost to maximum-cost
  End-if
  Move 'AARDVARK' to animal-type
else
  perform item-check
End-if
Move animal-type to print-animal-type.
Stop run.
```



# Initialize

---

- ▶ Another new statement in COBOL II was INITIALIZE - a quick way to set a lot of data items to an initial value
  - Numeric items go to zero, everything else to spaces
  - Useful to initialize a table, instead of using a loop
  - Since you can initialize a table with VALUE clauses at the elementary level, the most useful application might be to re-initialize the values for a table for each new use

# Initialize, 2

---

## ► An example:

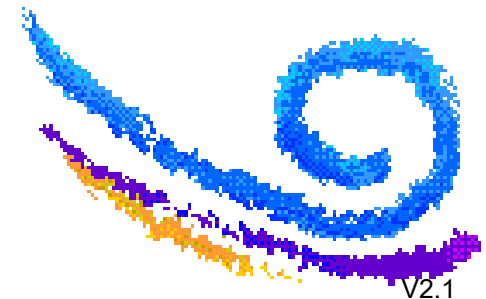
```
01 Customer-rec.  
  05 Customer-number pic x(8).  
  .  
  .  
  .  
  05 Invoices occurs 7 times.  
    10 Invoice-date pic 9(8).  
    10 Invoice-referenc pic x(5).  
    10 Invoice-balance pic s9(7)v99 packed-decimal.  
  .  
  .  
  .
```

In the procedure division, code:

```
initialize invoices
```

or

```
initialize customer-rec
```

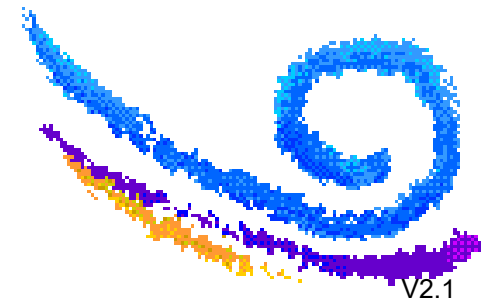


# CALL - passing arguments

- ▶ Between COBOL II and COBOL 3 (LE COBOL), new ways of passing (and receiving) arguments have evolved
- ▶ Classically, you coded:

`CALL 'routine' USING item1, item2, ...`

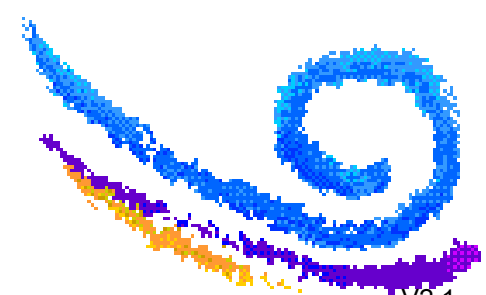
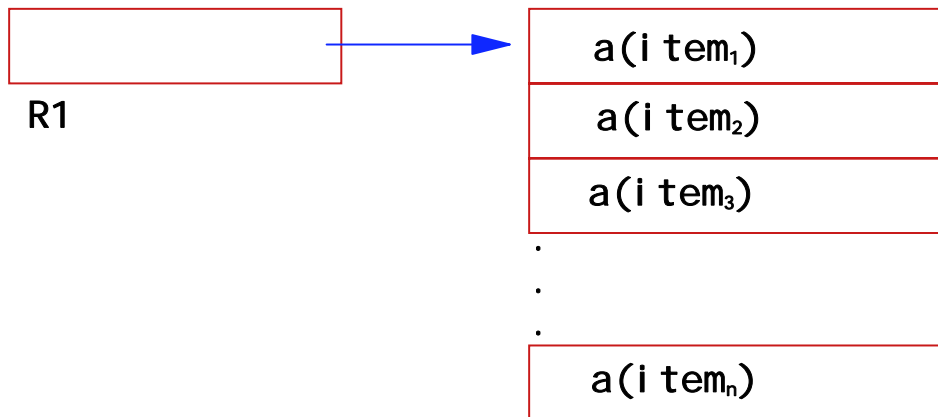
and behind the scenes, COBOL set up a list of addresses of the arguments, with R1 pointing to the list



# CALL - passing arguments, 2

---

## ▶ Classic argument format:



# CALL - passing arguments, 3

---

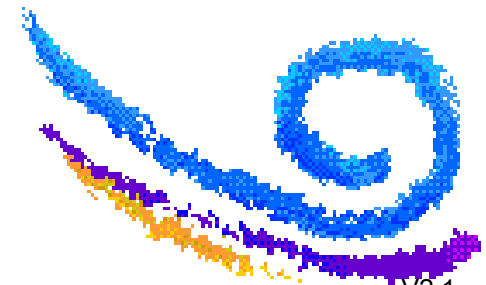
- ▶ The first change was to allow passing an argument by content - which means passing the address of a copy of the item
  - This allows passing literals in the argument string
  - And it also ensures if the called routine changed an argument, it only changed the copy, not the original data - protecting the data item's value from change
- ▶ Of course, then we had to have a way to describe the original style, so that became "by reference" and is the default so old code compiles correctly

# CALL - passing arguments, 4

---

- ▶ You can mix and match how you pass arguments: once you specify "by content" that is in effect for all subsequent arguments until you switch back by coding "by reference":

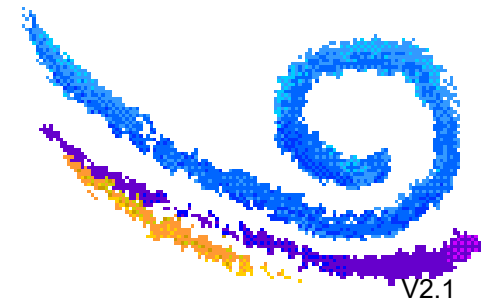
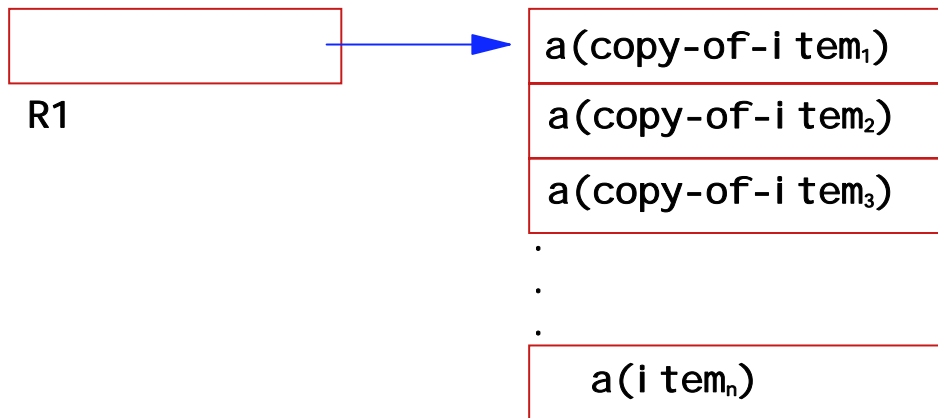
```
CALL 'CALC1' USING
  by content  'City tax',
              base-amt,
              city-rate,
  by reference city-tax,
              total-tax
```



# CALL - passing arguments, 5

---

- ▶ Mixed by content and by reference arguments:



# CALL - passing arguments, 6

---

- ▶ The next change was to allow passing an argument by value - which means passing a copy of the item's value right in the list of addresses!
  - Like by reference and by content, by value is transitive (applies to subsequent arguments until changed), and you can mix and match all three styles
- ▶ Also, not all data types can be passed by value: only binary, float, pointer, and single characters

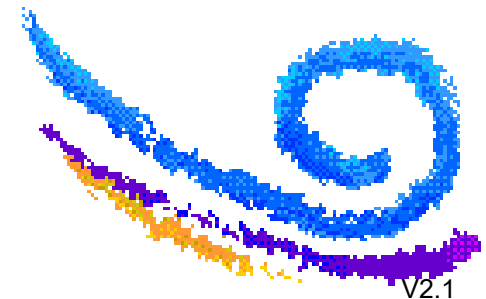


# CALL - passing arguments, 7

---

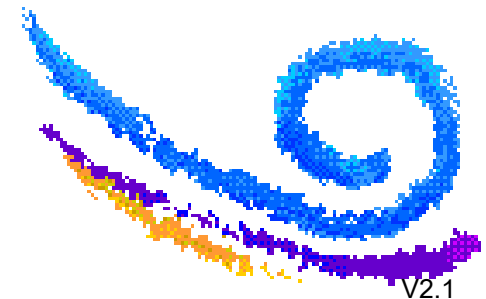
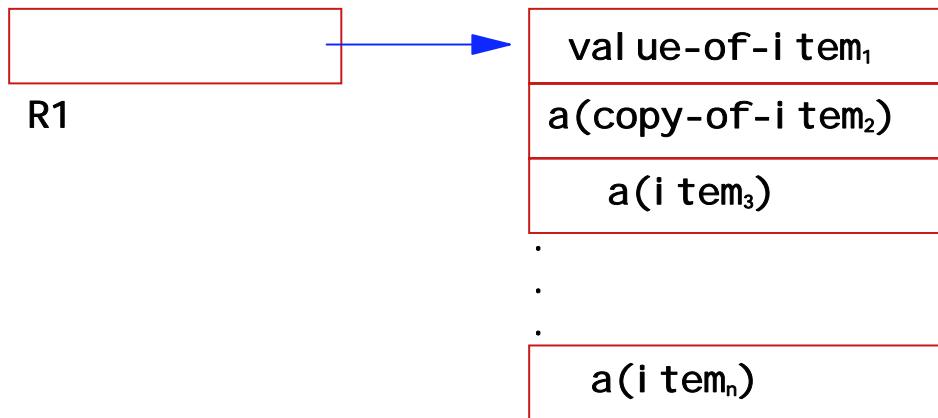
- ▶ Here's a mixed bag of arguments:

```
CALL 'CALCA' USING
  by value      use-count,
                max-count,
  by content    'City tax',
  by reference  base-amt,
                city-rate,
  by content    state-rate,
  by reference  total-tax
```



# CALL - passing arguments, 8

- ▶ Mixed by value, by content, and by reference arguments:



# CALL - passing arguments, 9

- ▶ Once you add **BY VALUE** in **CALL**, however, you need to add it to the **PROCEDURE DIVISION** header for its parameters
- ▶ A subroutine can't tell the difference between items passed **BY REFERENCE** and **BY CONTENT**: they are both pointers to data; the subroutine can't tell if it's the original or a copy
- ▶ But for **BY VALUE**, the subroutine needs to know it will find a value and not an address



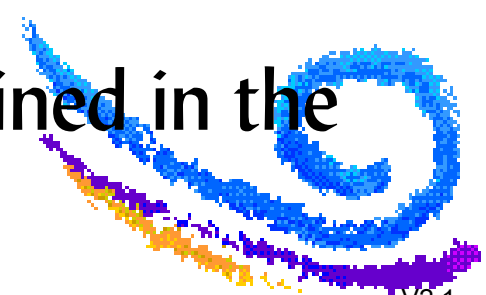
# CALL - passing arguments, 10

---

- ▶ For example:

```
PROCEDURE DIVISION USING
  by value      use-count,
                max-count,
  by reference  tax-type,
                base-amt,
                city-rate,
                state-rate,
                total-tax
```

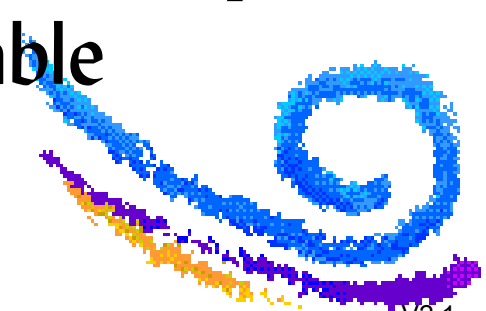
- ▶ Note that there are no by content specifications here: by content is not allowed in the procedure division header
- ▶ Of course, all the items here have to be defined in the linkage section



# CALL - passing arguments, I I

## ► WHY?

- To make it easier to call between COBOL and C!
- C thinks of arguments being passed on a stack, and a subroutine is expected to pop them off the stack
  - But z Series machines don't have [that kind of] a stack, so "by value" provides a reasonable simulation of stack behavior!



# CALL - RETURNING

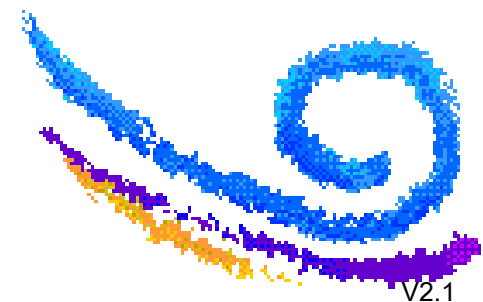
---

- ▶ One other change to CALL: a CALL statement can have a RETURNING phrase:

CALL '*routine*' USING ... RETURNING *data-item*

- ▶ *data-item* can be an elementary item or a group item
- ▶ The called *routine* must have a similar phrase in its procedure division header:

PROCEDURE DIVISION USING ... RETURNING *data-item*



# CALL - RETURNING, 2

---

- ▶ When a subroutine and its caller are set up to use RETURNING:
  - The subroutine must place a value in the named data-item before returning
  - The caller can use the data-item as any other item
  - The caller should not count on RETURN-CODE to be meaningful



# CALL - RETURNING, 3

---

## ► WHY?

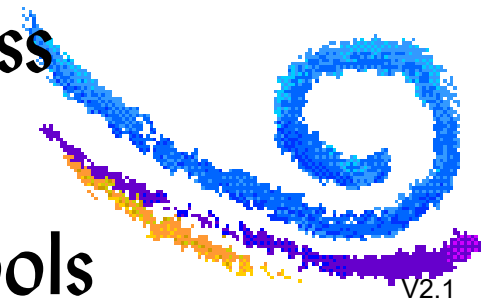
- Because now a COBOL program can call a C function directly! No need to code a complete C subroutine to use some of the neat facilities available
- Secondly, a COBOL program can look like a function to a C caller
- Note that COBOL to COBOL calls can use this facility too



# Language Environment

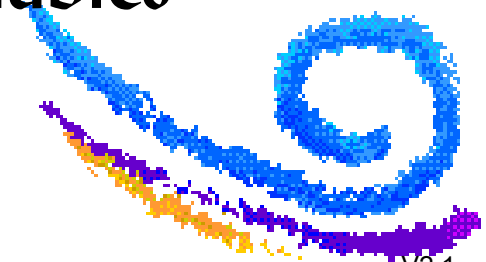
---

- ▶ Beginning with COBOL/370, all COBOL, PL/I, and C compilers use a common runtime package: Language Environment (LE)
- ▶ Although LE has gotten some bad press, it is overall a good thing:
  - Simplifies inter-language communication
  - Provides a suite of useful callable routines
  - Provides common code for mathematical functions and the like: more consistent results across languages
  - Common dump format and debugging tools



## Language Environment, 2

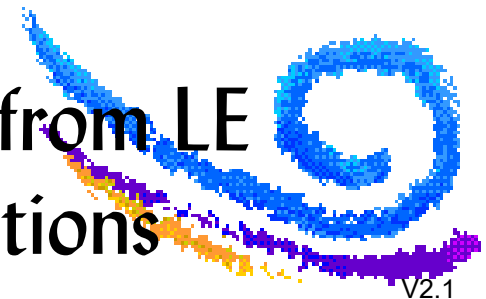
- ▶ We won't discuss LE in detail here (different course, more money), but LE provides several capabilities that are especially useful for the COBOL programmer
  - The ability to dynamically acquire storage outside your program without need for an intermediate Assembler subroutine
  - The ability to work with environment variables
  - [Indirect] support of intrinsic functions



# Intrinsic Functions

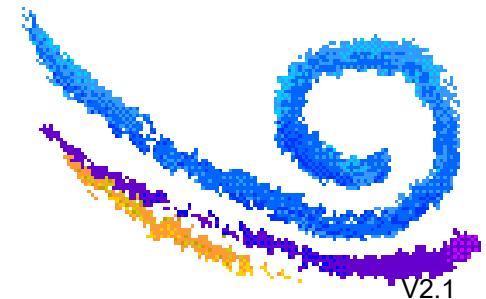
---

- ▶ In 1989, the ISO did something they had never done before: they amended an existing standard
  - Primarily because the new standard was way late
- ▶ IBM took the opportunity of this amendment to start the next line of COBOL compilers, the LE versions (sometimes called "COBOL 3" informally)
  - The main enhancement to COBOL, aside from LE support, was the addition of intrinsic functions



## Intrinsic Functions, 2

- ▶ By now, there are over 50 intrinsic functions supported in the current COBOL compilers
- ▶ "Intrinsic" because they are part of the language, not an add on
- ▶ "Functions" because they are invoked like a function: they are passed zero to three arguments and return a single value



# Intrinsic Functions, 3

---

- ▶ Here's a laundry list of the functions - you can get a general idea which ones might be useful in your work:

ACOS

ANNUITY

ASIN

ATAN

CHAR

COS

CURRENT-DATE

DATE-OF-INTEGER

DATE-TO-YYYYMMDD

DATEVAL

DAY-OF-INTEGER

DAY-TO-YYYYDDD

DISPLAY-OF

FACTORIAL

INTEGER

INTEGER-OF-DATE

INTEGER-OF-DAY

INTEGER-PART

LENGTH

LOG

LOG10

LOWER-CASE

MAX

MEAN

MEDIAN

MIDRANGE

MIN

MOD

NATIONAL-OF

NUMVAL

NUMVAL-C

ORD

ORD-MAX

ORD-MIN

PRESENT-VALUE

RANDOM

RANGE

REM

REVERSE

SIN

SQRT

STANDARD-DEVIATION

SUM

TAN

UNDATE

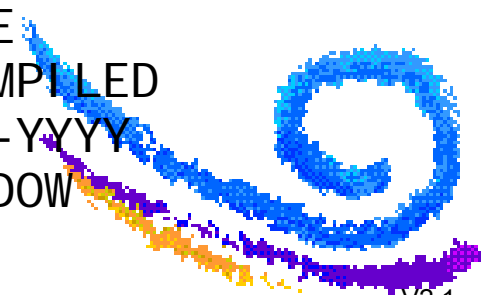
UPPER-CASE

VARIANCE

WHEN-COMPILED

YEAR-TO-YYYY

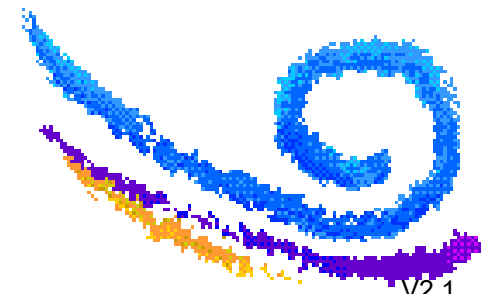
YEARWINDOW



# Tricks Using Modern COBOL

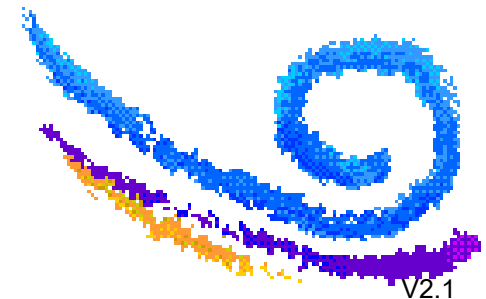
---

- ▶ Here's some things you couldn't do in COBOL before COBOL 3:
  - Dynamically allocate storage outside your program
  - Dynamically allocate a file from a COBOL program!
  - Create, set, change, access values in environment variables



# Tricks Using Modern COBOL, 2

- ▶ Here's some things you couldn't do in COBOL before COBOL 3:
  - Call z/OS UNIX kernel services directly
  - Call C functions directly
  - Create and use DLLs (Dynamic Link Libraries)
  - Take a snapshot dump and keep running



# Tricks Using Modern COBOL, 3

---

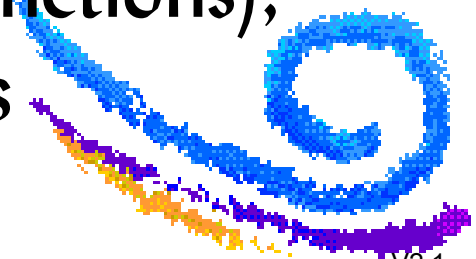
- ▶ Here's some things you couldn't do in COBOL before COBOL 3:
  - Use the DB2 coprocessor
  - Access files in the Hierarchical File System (HFS)
  - Code CGI programs to handle Web requests
  - Code your own condition handlers without having to use an Assembler ESTAE routine



# Tricks Using Modern COBOL, 4

---

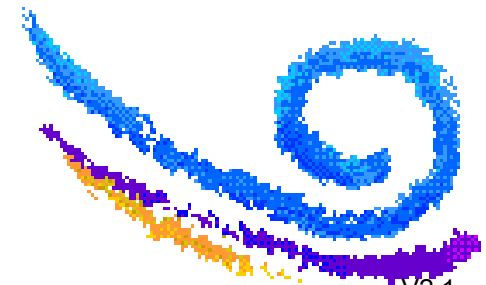
- ▶ Here's some things you couldn't do in COBOL before COBOL 3:
  - Code recursive programs
  - Code programs with multiple currency symbols, including the Euro
  - Use ALL as a subscript (certain intrinsic functions), simplifying table work in certain situations



# Debugging

---

- ▶ In COBOL II, there's a compile time option called FDUMP
  - If a program compiled with this option abended, it would produce a formatted dump of working storage: each data item by name and it's value
  - This greatly simplifies locating data items for debugging
  - At the cost of much larger object modules



## Debugging, 2

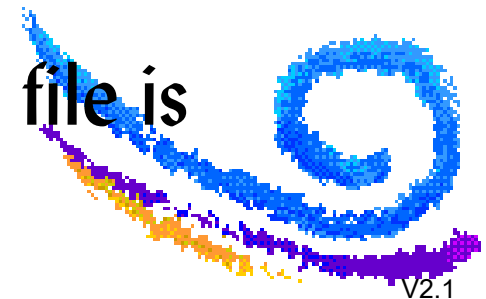
---

- ▶ COBOL 3 (COBOL/370 and its follow-ons) removed FDUMP but provided a new alternative - TEST
- ▶ TEST originally had two parameters:
  - `SYM | NOSYM` - embed (don't embed) the symbol table in the object code, like FDUMP used to do
  - `{NONE | STMT | PATH | BLOCK | ALL}` - indicating the program should be compiled to allow the debug tool to have "hooks" at certain points in the program

## Debugging, 3

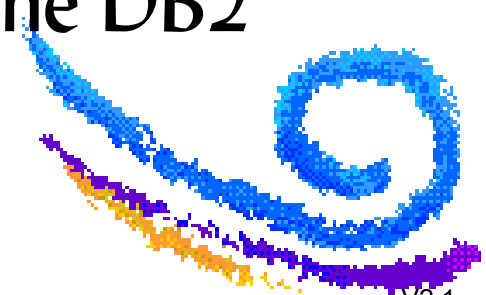
---

- ▶ In COBOL for OS/390 & VM (V2R2 of the new compilers), an additional suboption was added to TEST:
  - {SEPARATE | NOSEPARATE} - if you requested SEPARATE, at compile time you supply a DD statement named SYSDEBUG and the symbol table is written out there
    - Keeping the object module small; the object module just contains the name of the file
    - If the program abends, the symbol table file is dynamically allocated and used



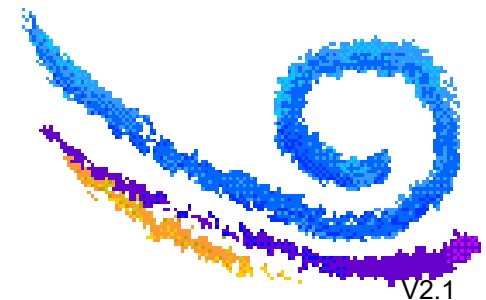
# Enterprise COBOL for z/OS ("COBOL 4")

- ▶ The Enterprise COBOL series of compilers have been designed to run under z/OS (as well as OS/390); the major enhancements are:
  - Unicode support
  - XML PARSE and XML GENERATE
  - Integrated CICS translator (analogous to the DB2 coprocessor)



## Enterprise COBOL for z/OS ("COBOL 4"), 2

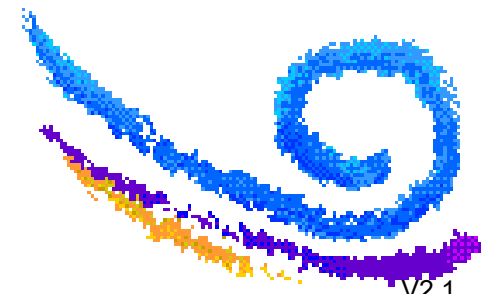
- ▶ The Enterprise COBOL series of compilers have been designed to run under z/OS (as well as OS/390); the major enhancements are:
  - Debugging enhancements
  - Limited multi-threading support
  - COBOL-Java interoperability



# COBOL 4 - Unicode Support

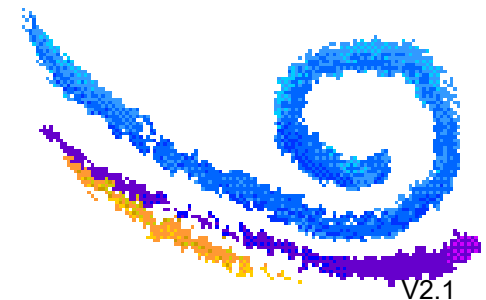
---

- ▶ Unicode support was introduced in V1R1 and extended in V4R1 of this compiler; major ways support is provided:
  - Unicode data type (National)
  - Unicode literals and Unicode hexadecimal literals
  - Automatic conversion between UTF-16 and EBCDIC for DISPLAY and ACCEPT



## COBOL 4 - Unicode Support, 2

- ▶ Unicode support was introduced in V1R1 and extended in V4R1 of this compiler; major ways support is provided:
  - Automatic conversion from EBCDIC to UTF-16 for MOVE when source is DISPLAY and target is EBCDIC
  - Intrinsic functions DISPLAY-OF and NATIONAL-OF to do explicit code page conversions



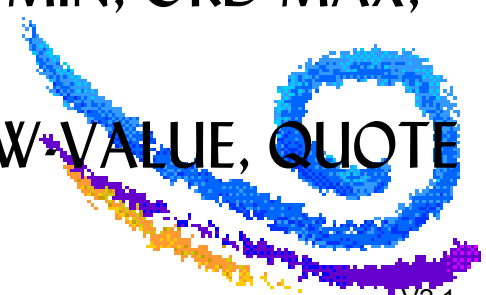
# COBOL 4 - Unicode Support, 3

---

► Unicode support was introduced in V1R1 and extended in V4R1 of this compiler; major ways support is provided:

■ Following verbs support Unicode data:

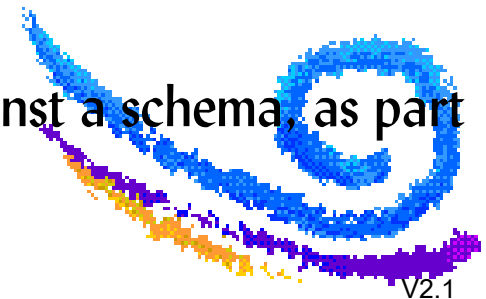
- SEARCH
- INSPECT
- STRING / UNSTRING
- DISPLAY
- EVALUATE
- Intrinsic functions: LOWER-CASE, UPPER-CASE, MAX, MIN, ORD-MAX, ORD-MIN, REVERSE
- Figurative constants: ZERO, SPACE, HIGH-VALUE, LOW-VALUE, QUOTE



# COBOL 4 - XML Support

---

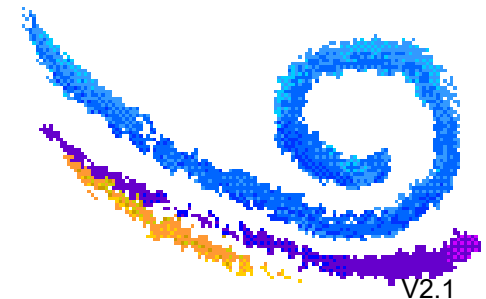
- ▶ XML support was introduced in V1R1 and extended in V3R3 and again in V4R1 and V4R2 of this compiler; major ways support is provided:
  - XML PARSE - given an input XML document, extract the field and attribute values into COBOL data types
  - XML GENERATE - given a COBOL structure, build an XML document
  - In V4R1, support was provided for many of the XML toolkit features to be available implicitly
  - In V4R2, support was added to validate an XML document against a schema, as part of the XML PARSE verb



# COBOL 4 - XML Support, 2

---

- ▶ XML support was introduced in V1R1 and extended in V3R3 and again in V4R1 and V4R2 of this compiler
  - This support allows you to take an existing transaction handling program (online or batch) and build a wrapper around it to handle XML style requests:
    - \* Convert incoming transaction from Unicode to EBCDIC - if needed
    - \* XML PARSE to build a COBOL style transaction from the XML style data
    - \* Call the transaction handler passing the COBOL style data
    - \* Accept the COBOL style results back
    - \* XML GENERATE to build an XML style response
    - \* Convert EBCDIC to Unicode - if needed
    - \* Transmit result back to sender



## COBOL 4 - Debugging - again

---

- ▶ In V4R1 of the Enterprise COBOL compiler, the TEST compiler option was revised as follows:

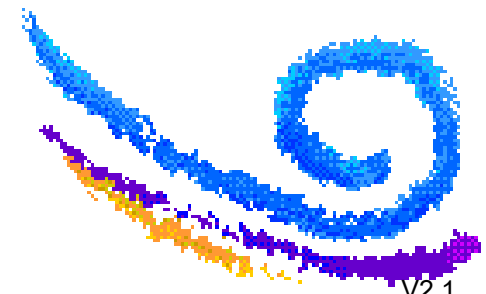
```
{NOTEST | TEST({HOOK | NOHOOK}  
                [, {NOSEPARATE | SEPARATE}]  
                [, {NOEJPD | EJPD})}) }
```

- ▶ The parameters may be specified in any order, and the meanings of the parameters are on the following pages

# COBOL 4 - Debugging - again, 2

---

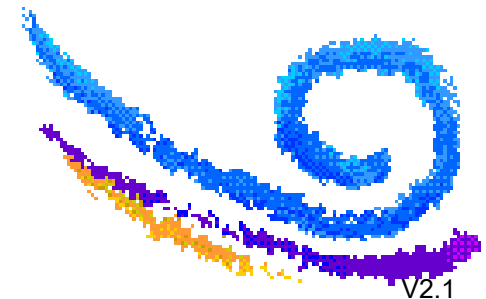
- ▶ **HOOK** causes compiled-in hooks to be generated at every statement, label, and path point (place where logic flow can change); **NOOPTIMIZE** is forced
- ▶ **NOHOOK** - no compiled-in hooks are generated; the Debug Tool dynamic debug facility may still be used
- ▶ **NOSEPARATE** - debugging information is embedded in the object code (**SYSDEBUG DD** statement points to target)
- ▶ **SEPARATE** - debugging information is stored in an external file, and the file name is embedded in the object code



# COBOL 4 - Debugging - again, 3

---

- ▶ NOEJPD - disables support for Debug Tool JUMPTO and GOTO commands, allowing normal optimization for OPTIMIZE option
- ▶ EJPD - enables support for Debug Tool JUMPTO and GOTO commands, at the price of reduced levels of optimization



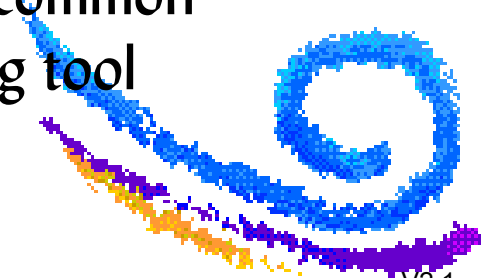
## COBOL 4 - Debugging - again, 4

- ▶ Notice the reference to "Debug Tool dynamic debug facility" two pages ago
- ▶ For some level of the Debug Tool, and some level of COBOL, even if you have not compiled with debug hooks, the Debug Tool can dynamically insert hooks when invoked
  - I'm not sure of the product levels because I've decided to write flawless code, so I have no need for debugging tools of any kind :-)



# Enterprise COBOL: A Summing Up

- ▶ Enterprise COBOL is the latest version of a continually improving product; it truly can help you with business applications in the modern world by:
  - Enhancing capabilities for existing code (larger data items, dynamic storage, enhanced CALL facilities, etc.)
  - Supporting more open technologies (such as XML, Unicode, DLLs, Java, z/OS UNIX, even CGI on a Web server)
  - Simplifying debugging in a complex environment (common dump format, symbol table format onabend, debug tool hooks)



# Enterprise COBOL - References

---

Enterprise COBOL IBM manuals found at:

[http://www.ibm.com/systems/z/os/zos/bkserv/zswpdf/#enterprise\\_cobol41](http://www.ibm.com/systems/z/os/zos/bkserv/zswpdf/#enterprise_cobol41)

COBOL standard

<http://www.cobolstandards.com/>

Unicode standard

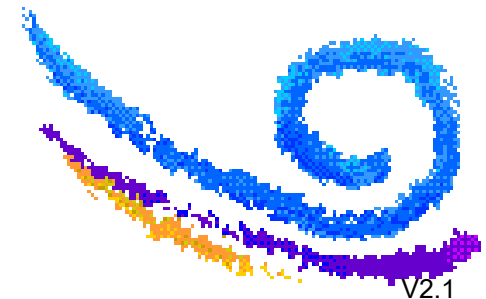
<http://www.unicode.org/>

World Wide Web Consortium (W3C)

<http://www.w3.org/> (follow links here to XML, HTML, etc.)

Training on COBOL and other technologies discussed here

<http://www.trainersfriend.com>





6790 East Cedar Avenue, Suite 201  
Denver, Colorado 80224  
USA

<http://www.trainersfriend.com>  
303.393.8716

Sales: [kitty@trainersfriend.com](mailto:kitty@trainersfriend.com)  
Technical: [steve@trainersfriend.com](mailto:steve@trainersfriend.com)