DSLGEN™ TOOLS DEMO

June, 2013

(Suppl. material & Speaker's notes July/Aug 2013)

Ted J. Biggerstaff

Software Generators, LLC

SOFTWARE GENERATORS, LLC™
Never Reprogram Again

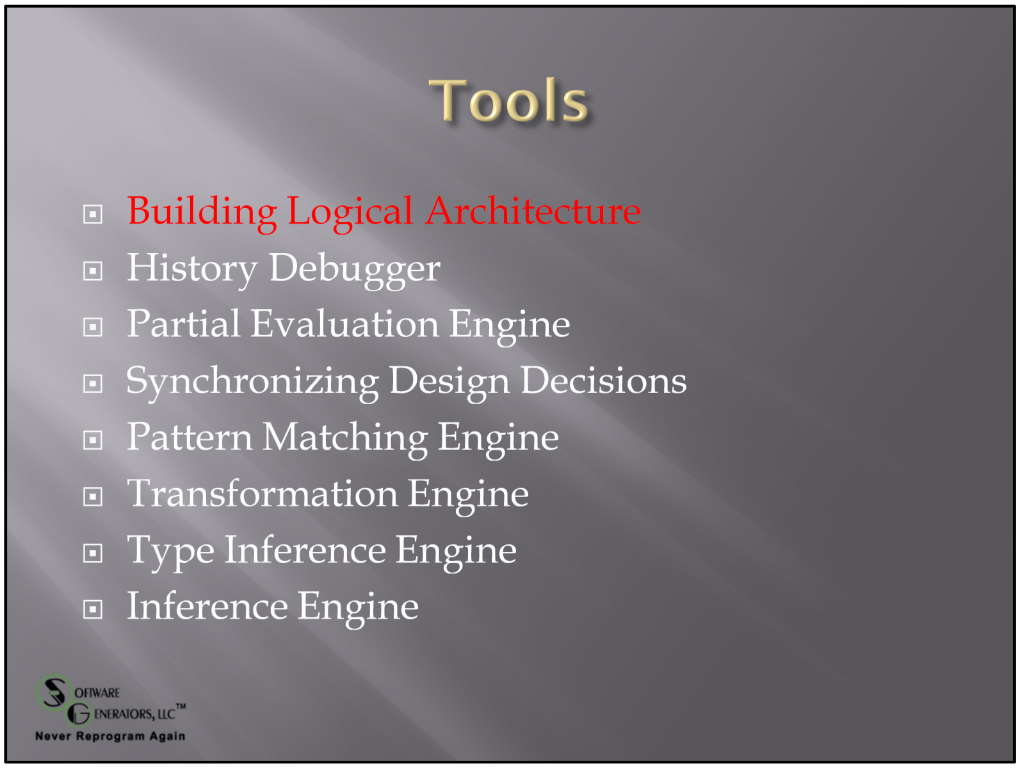
This presentation is supplemental to the previous overview presentation and assumes that the audience has understood the basic ideas from that presentation.

SOFTWARE GENERATORS, LLC™

Never Reprogram Again

SOFTWARE GENERATORS, LLC™

Never Reprogram Again

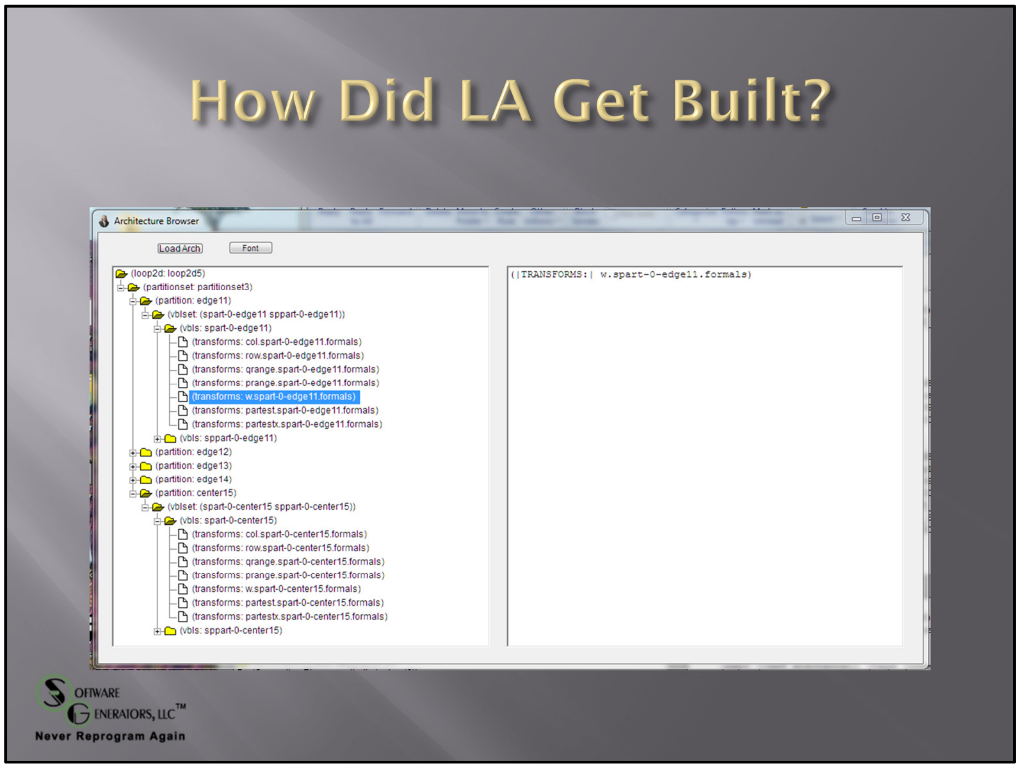


Tools

- Building Logical Architecture
- History Debugger
- Partial Evaluation Engine
- Synchronizing Design Decisions
- Pattern Matching Engine
- Transformation Engine
- Type Inference Engine
- Inference Engine

SOFTWARE GENERATORS, LLC™
Never Reprogram Again

Recall the discussion of the building and use of the logical architecture associated with a partially translated Implementation Neutral Specification (INS) of a computation. The next section will illustrate how such an LA gets constructed.

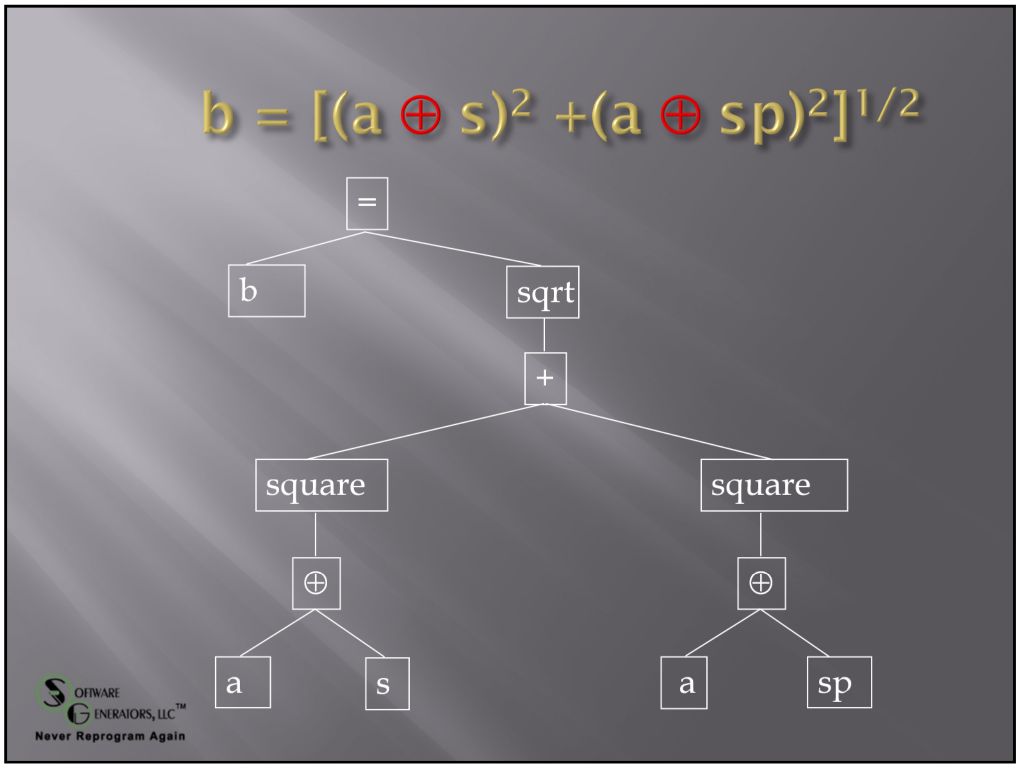


How Did LA Get Built?

```
(loop2d: loop2d5)
  (partitionset: partitionset3)
    (partition: edge11)
      (vblset: (spart-0-edge11 sppart-0-edge11))
        (vbls: spart-0-edge11)
          (transforms: col.spart-0-edge11.formals)
          (transforms: row.spart-0-edge11.formals)
          (transforms: qrange.spart-0-edge11.formals)
          (transforms: prange.spart-0-edge11.formals)
          (transforms: w.spart-0-edge11.formals)
          (transforms: partest.spart-0-edge11.formals)
          (transforms: partestx.spart-0-edge11.formals)
        (vbls: sppart-0-edge11)
    (partition: edge12)
    (partition: edge13)
    (partition: edge14)
    (partition: center15)
      (vblset: (spart-0-center15 sppart-0-center15))
        (vbls: spart-0-center15)
          (transforms: col.spart-0-center15.formals)
          (transforms: row.spart-0-center15.formals)
          (transforms: qrange.spart-0-center15.formals)
          (transforms: prange.spart-0-center15.formals)
          (transforms: w.spart-0-center15.formals)
          (transforms: partest.spart-0-center15.formals)
          (transforms: partestx.spart-0-center15.formals)
        (vbls: sppart-0-center15)
```

```
(|TRANSFORMS:| w.spart-0-edge11.formals)
```

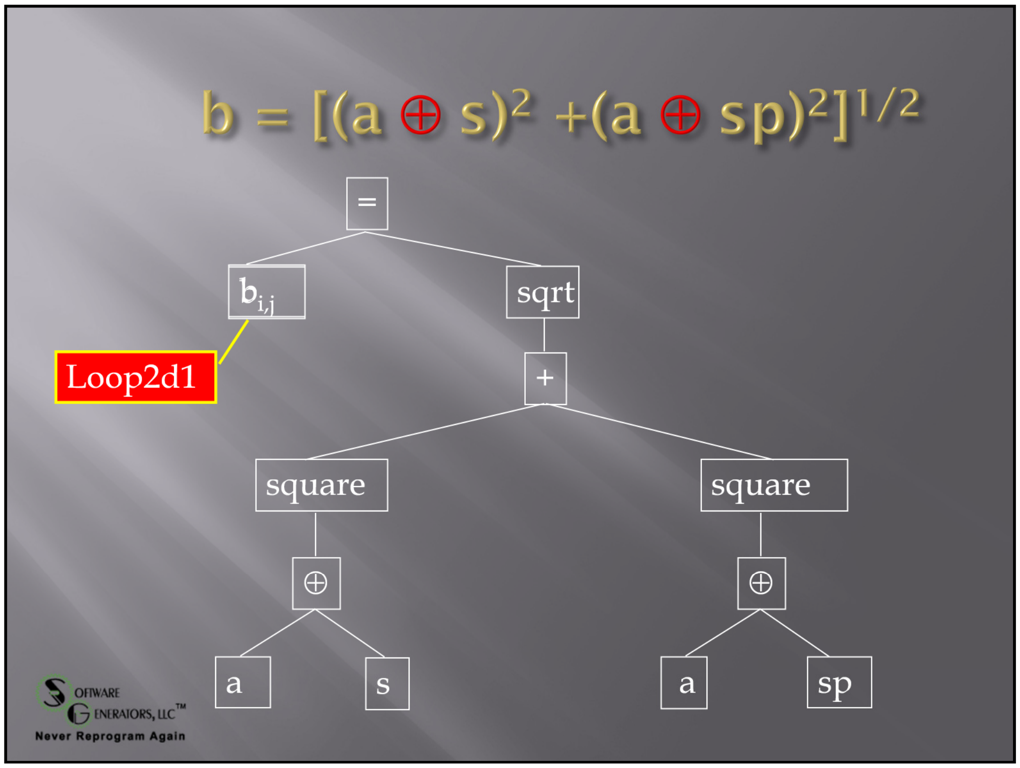
Recall the LA example from the overview presentation (illustrated by this Architecture Browser screen shot). How did this LA get built? The next (build) slide will animate the building of an LA from a fairly abstract and conceptual perspective.



$b = [(a \oplus s)^2 + (a \oplus sp)^2]^{1/2}$

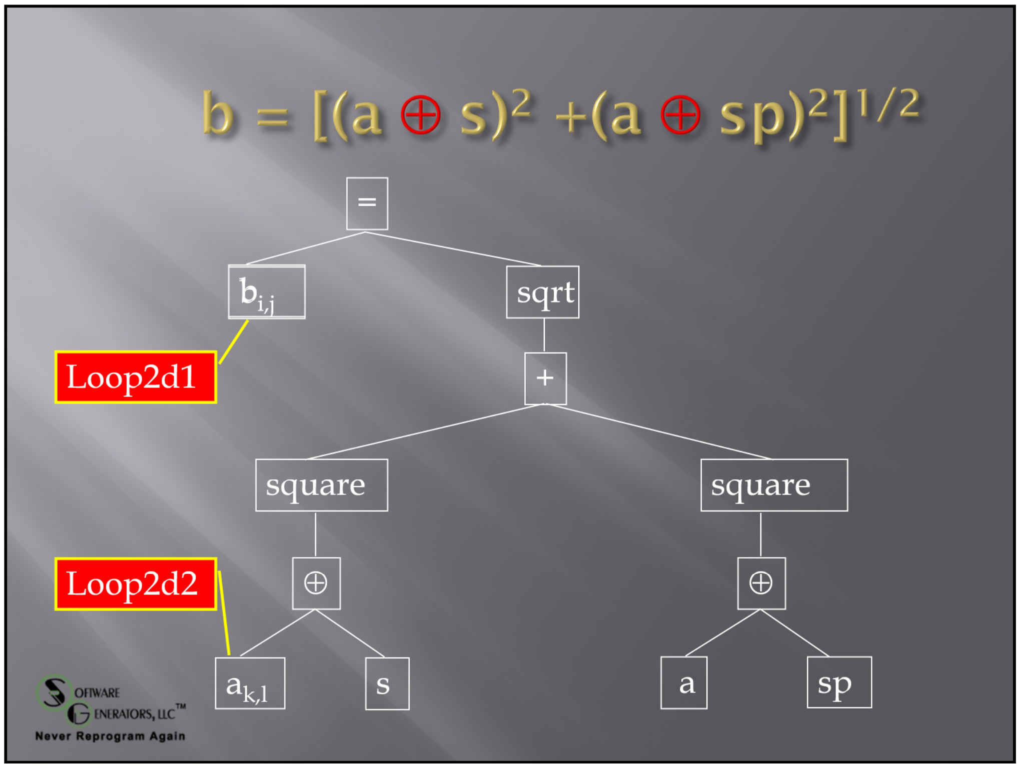
$$\begin{tikzcd}[column sep=small]
 & & = \arrow[dll, no head] \arrow[dr, no head] & & & \\
b & & & \mathrm{sqrt} \arrow[d, no head] & & \\
 & & & + \arrow[dll, no head] \arrow[drr, no head] & & \\
 & \mathrm{square} \arrow[d, no head] & & & & \mathrm{square} \arrow[d, no head] \\
 & \oplus \arrow[dl, no head] \arrow[dr, no head] & & & & \oplus \arrow[dl, no head] \arrow[dr, no head] \\
a & & s & & a & & sp
\end{tikzcd}$$

This hidden slide is the starting slide for a simulation of the **build slide** shown at the end of this series. These hidden slides are provided to clarify the individual build actions for readers of the pdf version of the Powerpoint presentation.



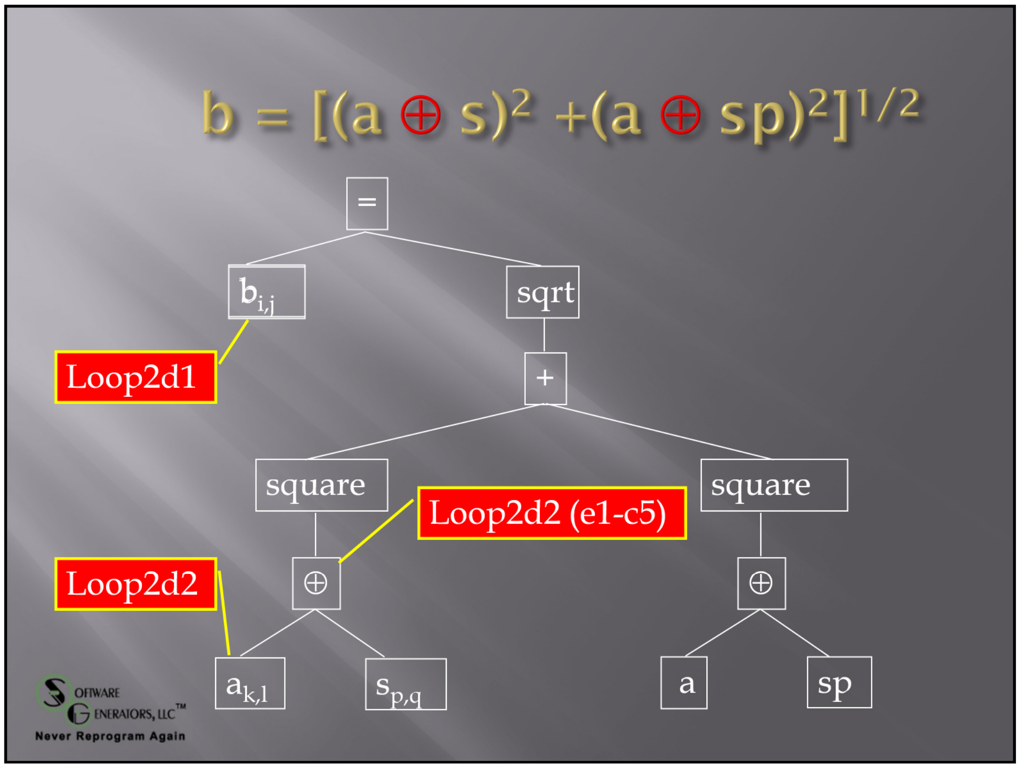
$b = [(a \oplus s)^2 + (a \oplus sp)^2]^{1/2}$

This hidden slide simulates (for pdf file readers) two steps of the build slide at the end of this series.



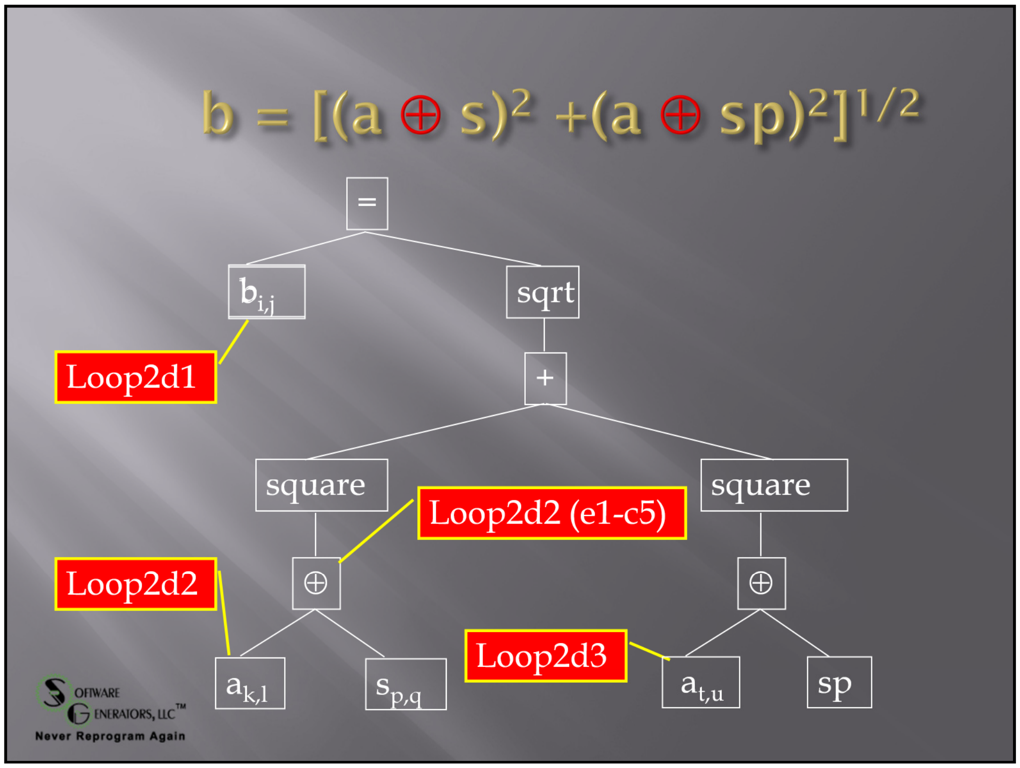
$b = [(a \oplus s)^2 + (a \oplus sp)^2]^{1/2}$

This hidden slide simulates (for pdf file readers) two steps of the build slide at the end of this series.



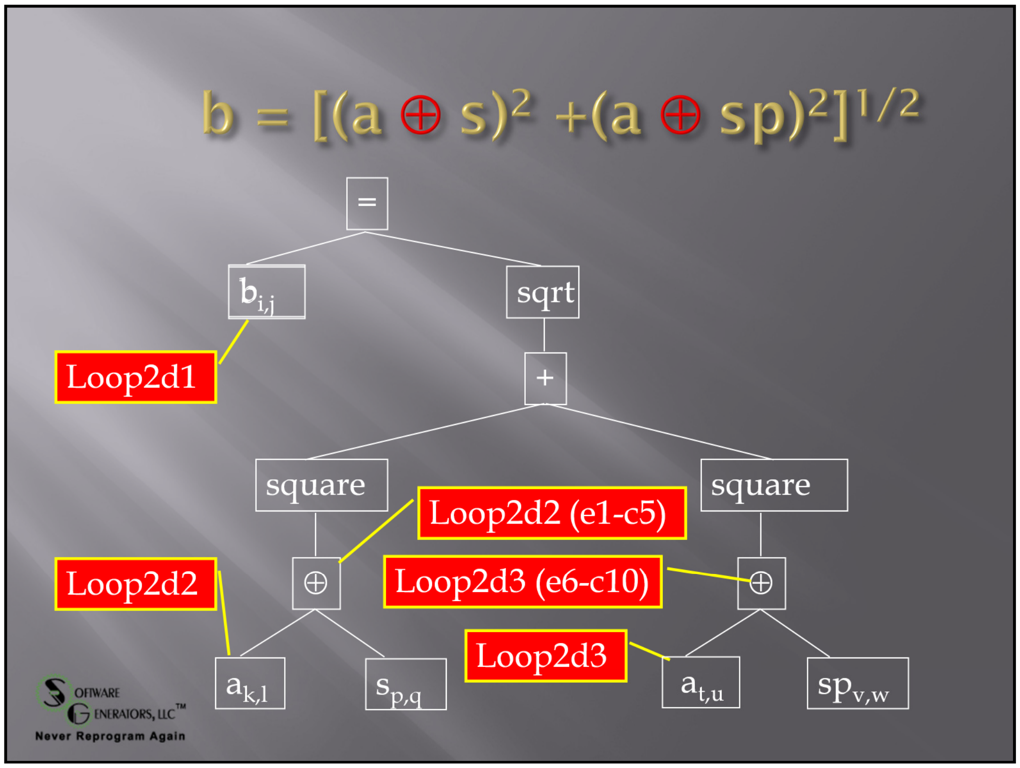
$b = [(a \oplus s)^2 + (a \oplus sp)^2]^{1/2}$

This hidden slide simulates (for pdf file readers) two steps of the build slide at the end of this series.



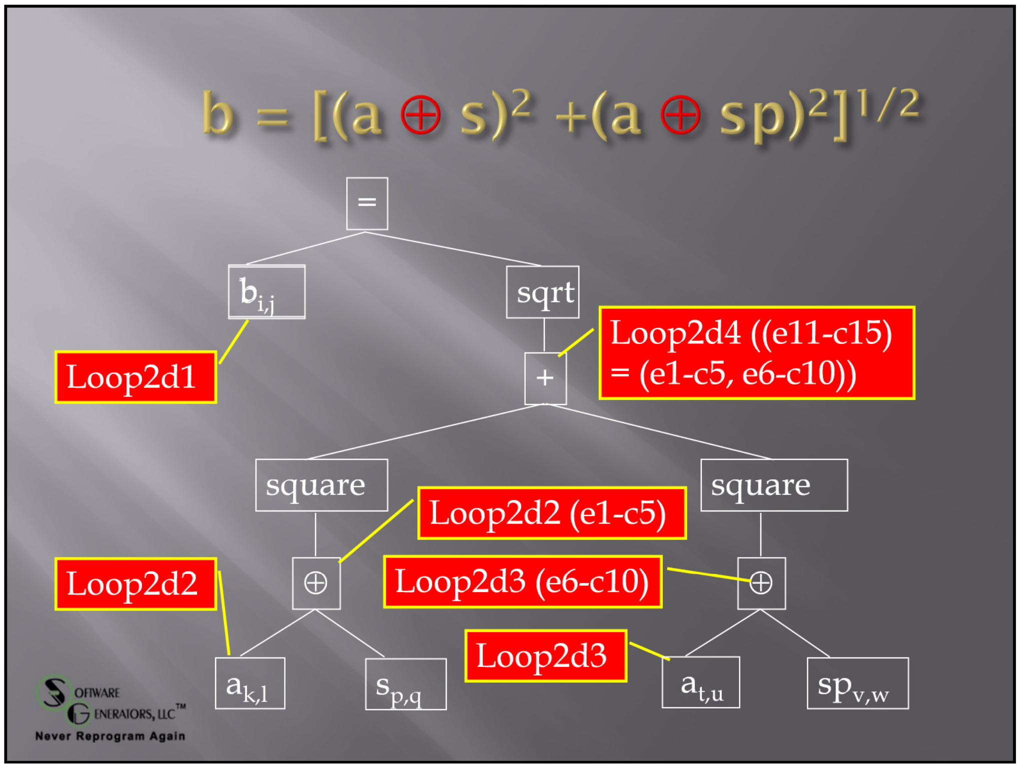
$b = [(a \oplus s)^2 + (a \oplus sp)^2]^{1/2}$

This hidden slide simulates (for pdf file readers) two steps of the build slide at the end of this series.



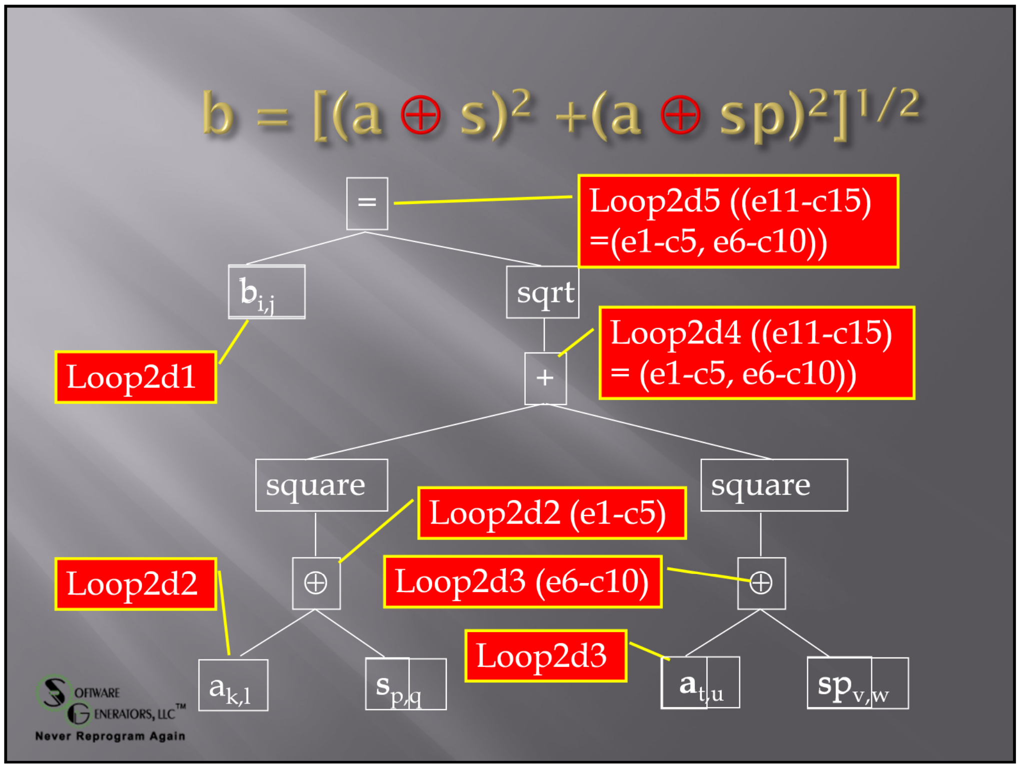
$b = [(a \oplus s)^2 + (a \oplus sp)^2]^{1/2}$

This hidden slide simulates (for pdf file readers) two steps of the build slide at the end of this series.

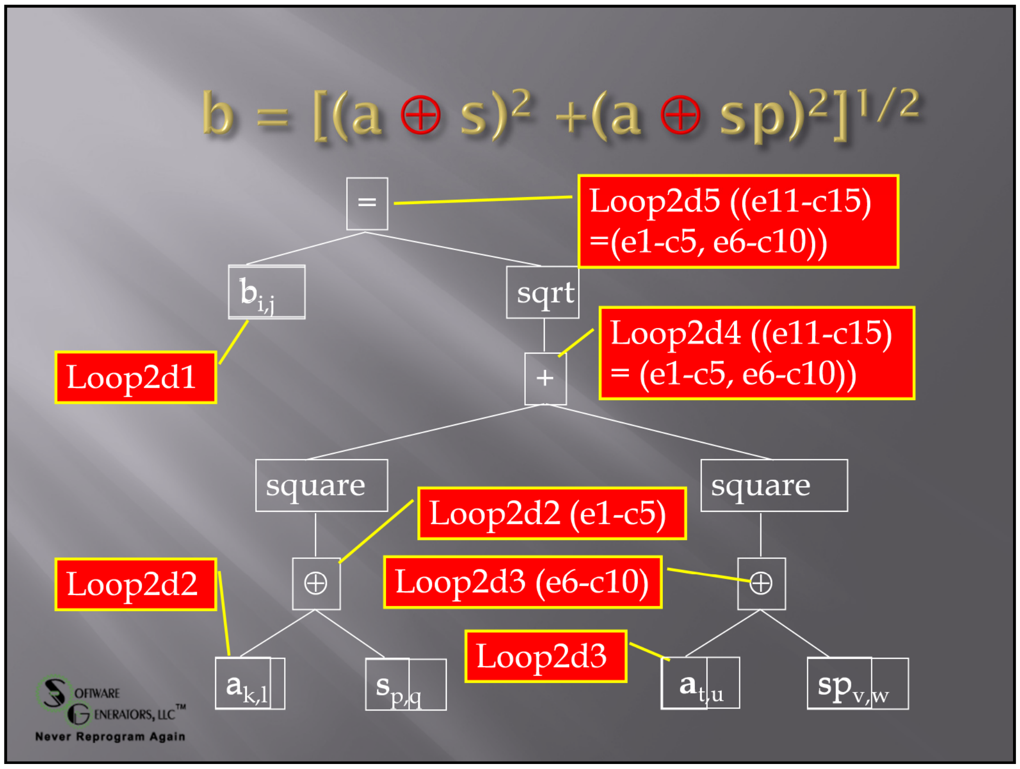


$b = [(a \oplus s)^2 + (a \oplus sp)^2]^{1/2}$

This hidden slide simulates (for pdf file readers) two steps of the build slide at the end of this series.


$b = [(a \oplus s)^2 + (a \oplus sp)^2]^{1/2}$
=
Loop2d5 ((e11-c15) =(e1-c5, e6-c10))
$b_{i,j}$
sqrt
Loop2d1
+
Loop2d4 ((e11-c15) = (e1-c5, e6-c10))
square
Loop2d2 (e1-c5)
square
Loop2d2
⊕
Loop2d3 (e6-c10)
⊕
Loop2d3
$a_{k,l}$
$s_{p,q}$
$a_{t,u}$
$sp_{v,w}$
SOFTWARE GENERATORS, LLC™
Never Reprogram Again

This hidden slide simulates (for pdf file readers) two steps of the build slide at the end of this series.



This is a build slide. 17 mouse clicks will animate the processing of this AST showing:

•the creation of loop APC (constraints),
•the introduction of loop index names (e.g., i and j, and k and l),
•the partial translation of the expression (e.g., b -> b[i,j]),
•the creation of partitions associated with loop APCs (e.g., "(e1-c5)"),
•the propagation of the APCs over the AST, and
•the merging of APCs.

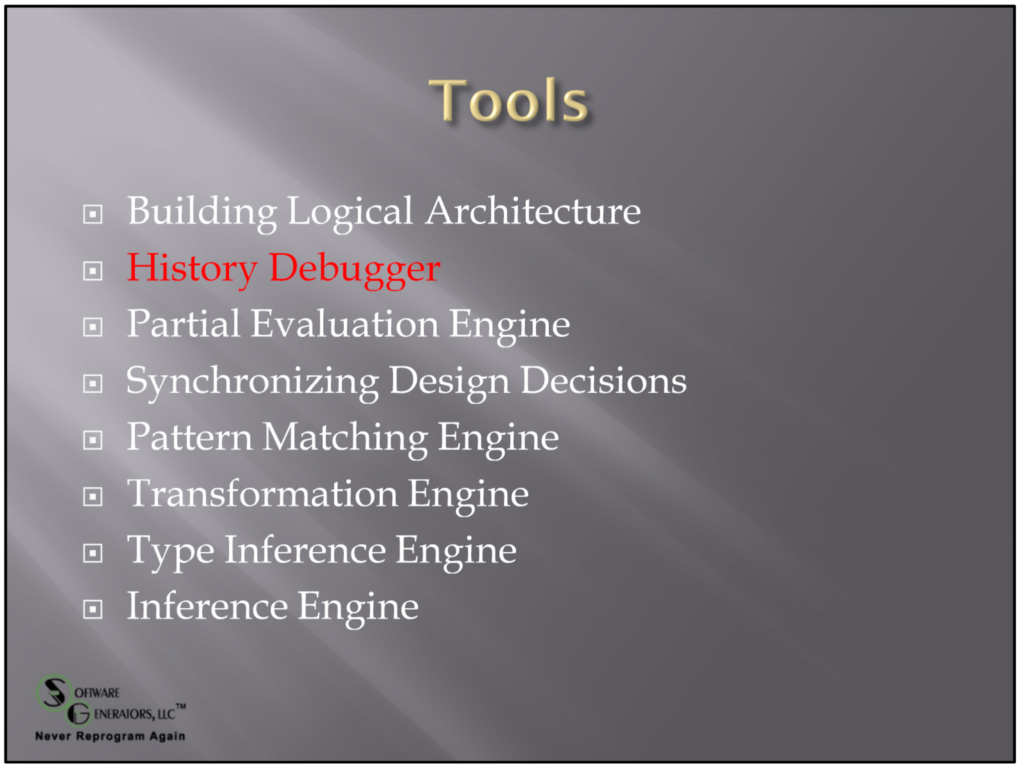
Bookmarks for **DSLGenICSRToolsDemoV4.dxl** demo file. (**NB:** For the **DSLGenICSRToolsDemoV4.dxl** example, the analogical correspondences between the presentation example and the History Debugger example are:

a corresponds to c,
b corresponds to d,
i corresponds to idx3,
j corresponds to idx4,
s corresponds to spart.
sp corresponds to sppart.)

Bookmarks for **DSLGenICSRToolsDemoV3.dxl** demo--

[Format: (***Bookmark name, (history outline node title, history outline node index)***)].

((Creation of Loop2d1 for D Image)) (partitioningcompositeleaf (4 5)))
(((Creation of Loop2d2 for C Image)) (partitioningcompositeleaf (4 6)))
(((Creation edge1)) (makeapartition (4 8 4 4)))
(((Creation of Loop2d3 for 2nd occurrence of C Image)) (partitioningcompositeleaf (4 18)))
(((Creation of edge6)) (makeapartition (4 20 4 4)))
(((Loop2d4 = loop2d2 + loop2d3)) (partitioningfunctionalopsonpixelcomposites (4 46)))
(((Loop2d5 = loop2d1 + loop2d4)) (partitioningfunctionalopsonpixelcomposites (4 67)))
(((PartesttX results for neighborhood w.sppart-0-edge11)) (adtcomponentredirection (10 7)))
(((Conv Op for neighborhood spart-0-edge11)) (partrightlinearproduct (10 14 1)))
(((Row Method-Transform for neighborhood spart-0-edge11)) (adtcomponentredirection (10 14 2)))
(((Col Method-Transform for neighborhood spart-0-edge11)) (adtcomponentredirection (10 14 3)))
(((W Method-Transform for neighborhood spart-0-edge11)) (adtcomponentredirection (10 14 4)))
((((Final inline result for Conv op of neighborhood w.spart-0-edge11)) (dsoperatorredirection (10 14)))
((Partially evaluate Conv Expr for neighborhood spart-0-edge11) (pe (11 7 1 2 3 3)))
((Square Root of (0 + 0), RHS of edge11 loop) (partitioningsimplifyletbodies (11 8)))
(((Example synch decision; Idx7 -> idx3)) (sprefine22640 (5 5 2 1 1)))

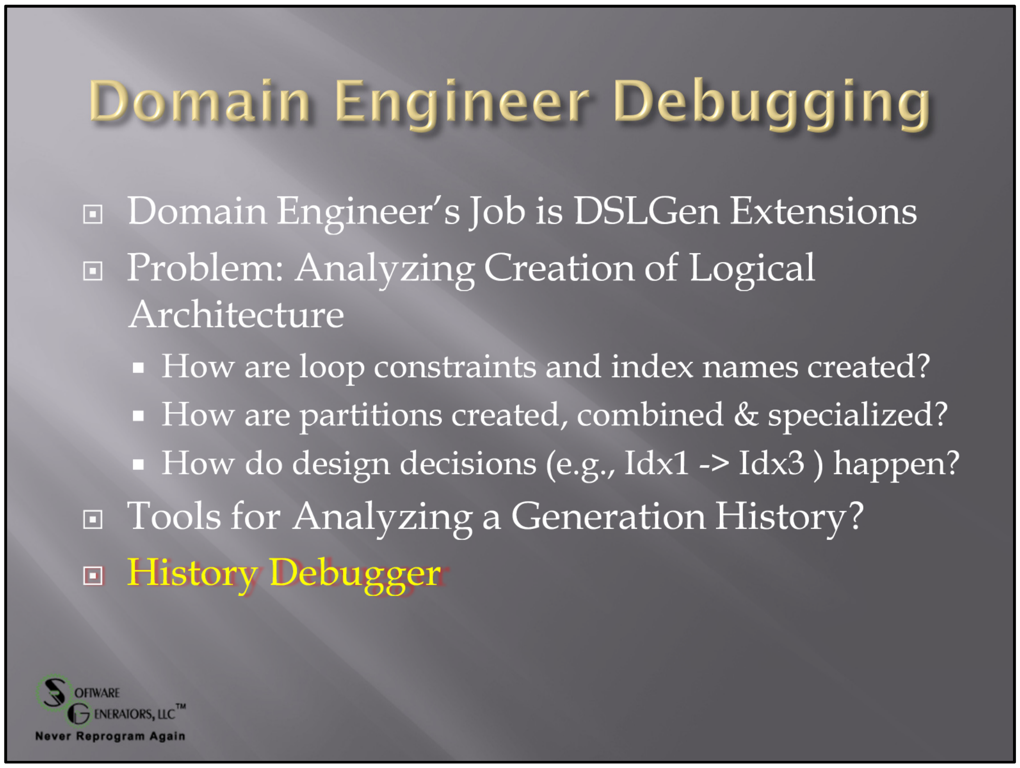


Tools

- Building Logical Architecture
- History Debugger
- Partial Evaluation Engine
- Synchronizing Design Decisions
- Pattern Matching Engine
- Transformation Engine
- Type Inference Engine
- Inference Engine

SOFTWARE GENERATORS, LLC™
Never Reprogram Again

The History Debugger is the key tool that the Domain Engineer uses to analyze and potentially debug a derivation history for some code generation.



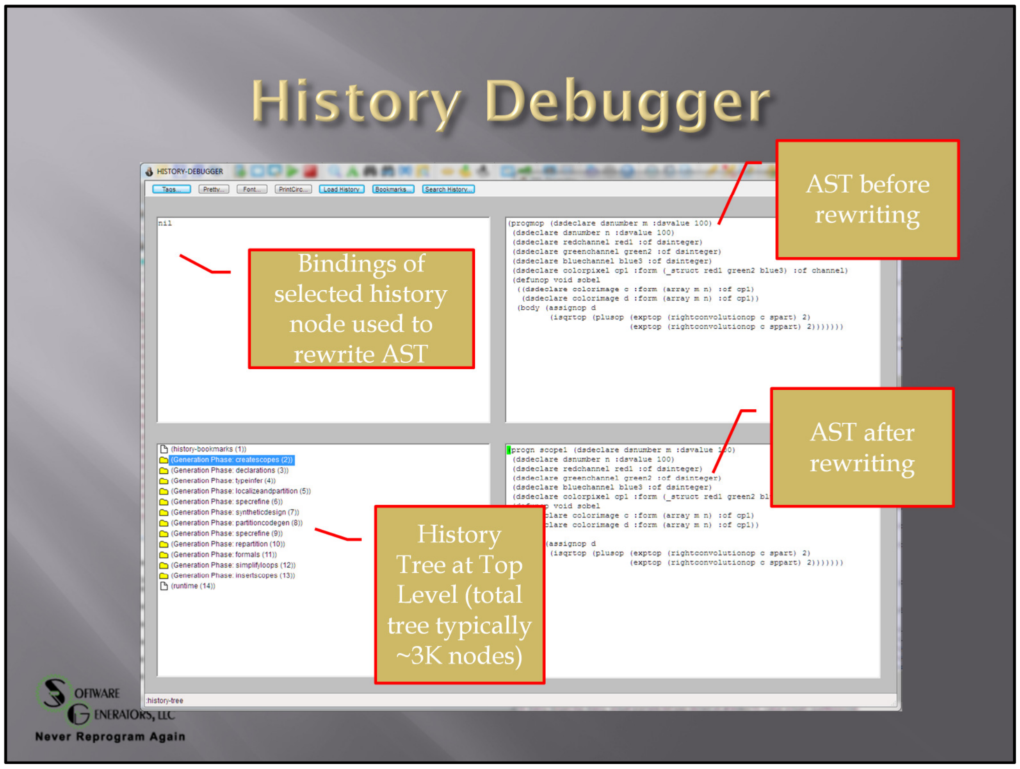
Domain Engineer Debugging

- Domain Engineer's Job is DSLGen Extensions
- Problem: Analyzing Creation of Logical Architecture
  - How are loop constraints and index names created?
  - How are partitions created, combined & specialized?
  - How do design decisions (e.g., Idx1 -> Idx3) happen?
- Tools for Analyzing a Generation History?
- History Debugger

SOFTWARE GENERATORS, LLC™
Never Reprogram Again

Rationalization of the existence of the History Debugger (HD). A key point is that the HD is **not** for the Application Programmer who operates with DSLGen™ as if it were just a really smart "compiler." The HD is designed for use by the Domain Engineer whose job is to improve and extend DSLGen™.

History Debugger



```
nil
```

```
(progmop (dsdeclare dsnumber m :dsvalue 100)
 (dsdeclare dsnumber n :dsvalue 100)
 (dsdeclare redchannel red1 :of dsinteger)
 (dsdeclare greenchannel green2 :of dsinteger)
 (dsdeclare bluechannel blue3 :of dsinteger)
 (dsdeclare colorpixel cp1 :form (_struct red1 green2 blue3) :of channel)
 (defunop void sobel
  ((dsdeclare colorimage c :form (array m n) :of cp1)
   (dsdeclare colorimage d :form (array m n) :of cp1))
  (body (assignop d
         (isqrtop (plusop (exptop (rightconvolutionop c spart) 2)
                          (exptop (rightconvolutionop c sppart) 2)))))))
```

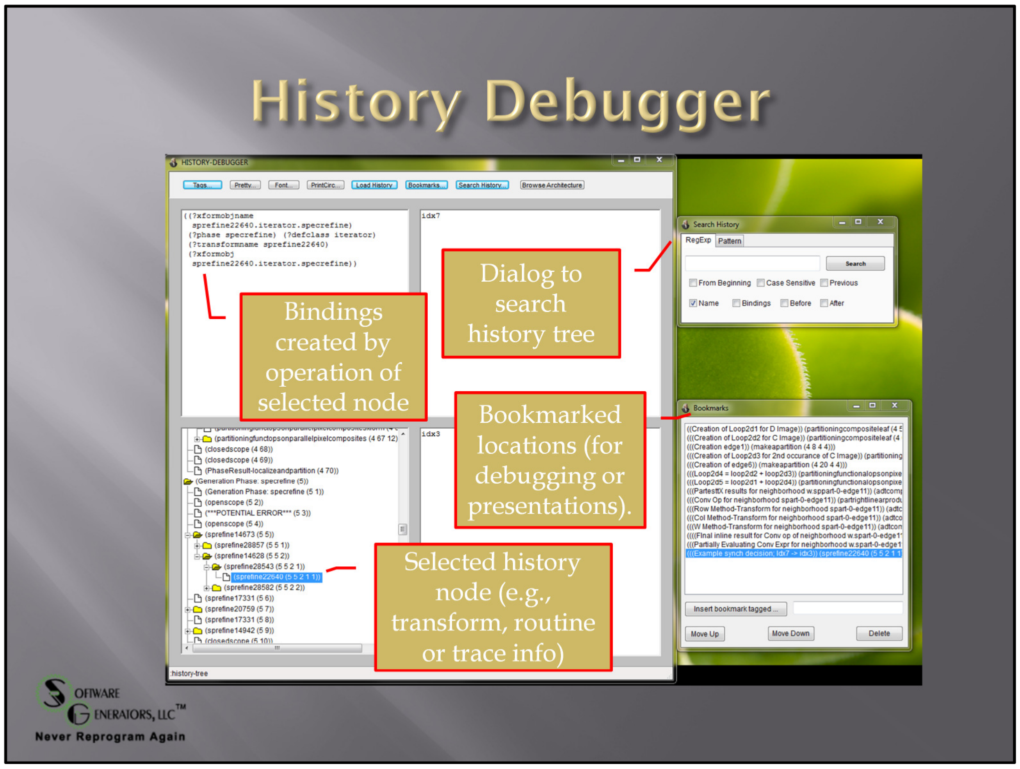
```
(history-bookmarks (1))
(Generation Phase: createscopes (2))
(Generation Phase: declarations (3))
(Generation Phase: typeinfer (4))
(Generation Phase: localizeandpartition (5))
(Generation Phase: specrefine (6))
(Generation Phase: syntheticdesign (7))
(Generation Phase: partitioncodegen (8))
(Generation Phase: specrefine (9))
(Generation Phase: repartition (10))
(Generation Phase: formals (11))
(Generation Phase: simplifyloops (12))
(Generation Phase: insertscopes (13))
(runtime (14))
```

```
(progn scope1 (dsdeclare dsnumber m :dsvalue 100)
 (dsdeclare dsnumber n :dsvalue 100)
 (dsdeclare redchannel red1 :of dsinteger)
 (dsdeclare greenchannel green2 :of dsinteger)
 (dsdeclare bluechannel blue3 :of dsinteger)
 (dsdeclare colorpixel cp1 :form (_struct red1 green2 bl
 p void sobel
 clare colorimage c :form (array m n) :of cp1)
 clare colorimage d :form (array m n) :of cp1))

 (assignop d
  (isqrtop (plusop (exptop (rightconvolutionop c spart) 2)
                   (exptop (rightconvolutionop c sppart) 2)))))))
```

Introduce the HD and describe the windows. The outline window (lower left) is a directory like view of the history tree where any node selected (i.e., some generator operation, either atomic or a composition of many sub-operations) in the outline determines what is shown in the three other windows. The upper left window contains variable bindings associated with the operation (e.g., the bindings resulting from a pattern matching operation associated with a transformation). The upper right window shows the Abstract Syntax Tree (AST) focus before the operation started and the lower right shows the AST after the operation has completed. In many instances, the bindings, before and after windows are used to display trace information and those cases do not really reflect a true generator transformation. DSLGen™ has 30 or 40 trace flags that can be turned to debug in particular areas of the generator (e.g., the partial evaluator). So, the size of the history tree can vary quite a bit.

The instance of the history outline shown in this slide is the top level summarization of the generator's operations for a particular case. At the top level, the outline items are mostly the names of generator's phases (e.g., process <u>scopes</u>, process <u>declarations</u>, <u>typeinfer</u>, <u>syntheticdesign</u>, etc.). The last outline item reports runtime in seconds (e.g., 75 seconds or so for examples like this one), the total number of nodes in the history tree (e.g., around 3,000 for this scale of problem) and the number of actual transformations (e.g., around 800). **(Speaker's note: if running a live example, show how the bindings, before and after windows change by selecting some lower level nodes within one of the phases.)**

History Debugger


HISTORY-DEBUGGER
Tags...
Pretty...
Font...
PrintCirc...
Load History
Bookmarks...
Search History...
Browse Architecture
Bindings created by operation of selected node
Dialog to search history tree
Bookmarked locations (for debugging or presentations).
Selected history node (e.g., transform, routine or trace info)
Search History
RegExp
Pattern
Search
Bookmarks
Insert bookmark tagged ...
Move Up
Move Down
Delete
:history-tree
SOFTWARE GENERATORS, LLC™
Never Reprogram Again

```
((?xformobjname
  sprefine22640.iterator.specrefine)
 (?phase specrefine) (?defclass iterator)
 (?transformname sprefine22640)
 (?xformobj
  sprefine22640.iterator.specrefine))
```

```
idx7
```

```
    (partitioningfunctopsonparallelpixelcomposites (4 67 12)
  (closedscope (4 68))
  (closedscope (4 69))
  (PhaseResult-localizeandpartition (4 70))
(Generation Phase: specrefine (5))
  (Generation Phase: specrefine (5 1))
  (openscope (5 2))
  (***POTENTIAL ERROR*** (5 3))
  (openscope (5 4))
  (sprefine14673 (5 5))
    (sprefine28857 (5 5 1))
    (sprefine14628 (5 5 2))
      (sprefine28543 (5 5 2 1))
        (sprefine22640 (5 5 2 1 1))
      (sprefine28582 (5 5 2 2))
  (sprefine17331 (5 6))
  (sprefine20759 (5 7))
  (sprefine17331 (5 8))
  (sprefine14942 (5 9))
  (closedscope (5 10))
```

```
idx3
```

[ ] From Beginning [ ] Case Sensitive [ ] Previous

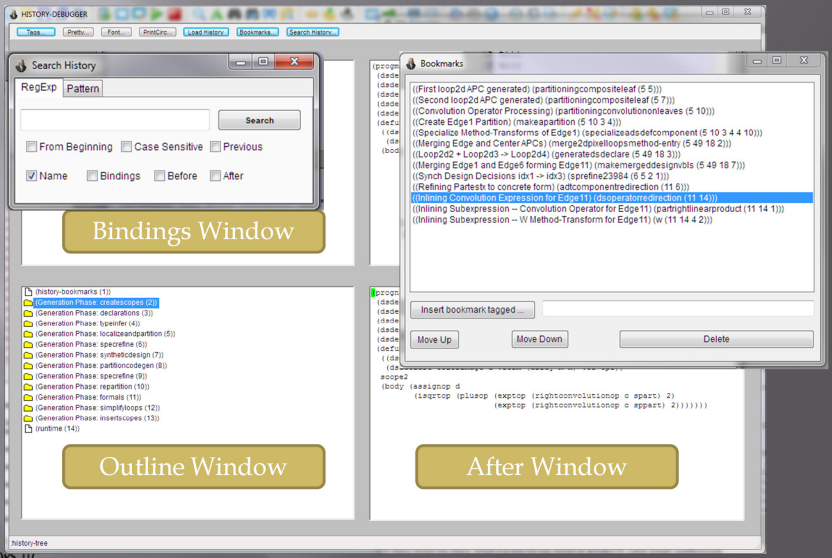
[✓ Name] [ ] Bindings [ ] Before [ ] After

```
((Creation of Loop2d1 for D Image)) (partitioningcompositeleaf (4 5
(((Creation of Loop2d2 for C Image)) (partitioningcompositeleaf (4
(((Creation edge1)) (makeapartition (4 8 4 4)))
(((Creation of Loop2d3 for 2nd occurance of C Image)) (partitioning
(((Creation of edge6)) (makeapartition (4 20 4 4)))
(((Loop2d4 = loop2d2 + loop2d3)) (partitioningfunctionalopsonpixe
(((Loop2d5 = loop2d1 + loop2d4)) (partitioningfunctionalopsonpixe
(((PartesttX results for neighborhood w.sppart-0-edge11)) (adtcom
(((Conv Op for neighborhood spart-0-edge11)) (partrightlinearprodu
(((Row Method-Transform for neighborhood spart-0-edge11)) (adtc
(((Col Method-Transform for neighborhood spart-0-edge11)) (adtco
(((W Method-Transform for neighborhood spart-0-edge11)) (adtcon
((((Final inline result for Conv op of neighborhood w.spart-0-edge1
(((Partially Evaluating Conv Expr for neighborhood w.spart-0-edge1
(((Example synch decision; Idx7 -> idx3)) (sprefine22640 (5 5 2 1 1
```

Describe the search dialog and the bookmarking dialog and explain why and how they are used. Why: finding specific operations or transformations is often like looking for a needle in a haystack. Search is a godsend in such cases. Bookmarks are great for debugging related but widely separated steps, as well as for use in presentations.

(Speaker's note: if running a live example, select a bookmark like the point where one of the loop constraints is created or the point of inlining of the definition of W of a neighborhood for a live demonstration.)

History Debugger

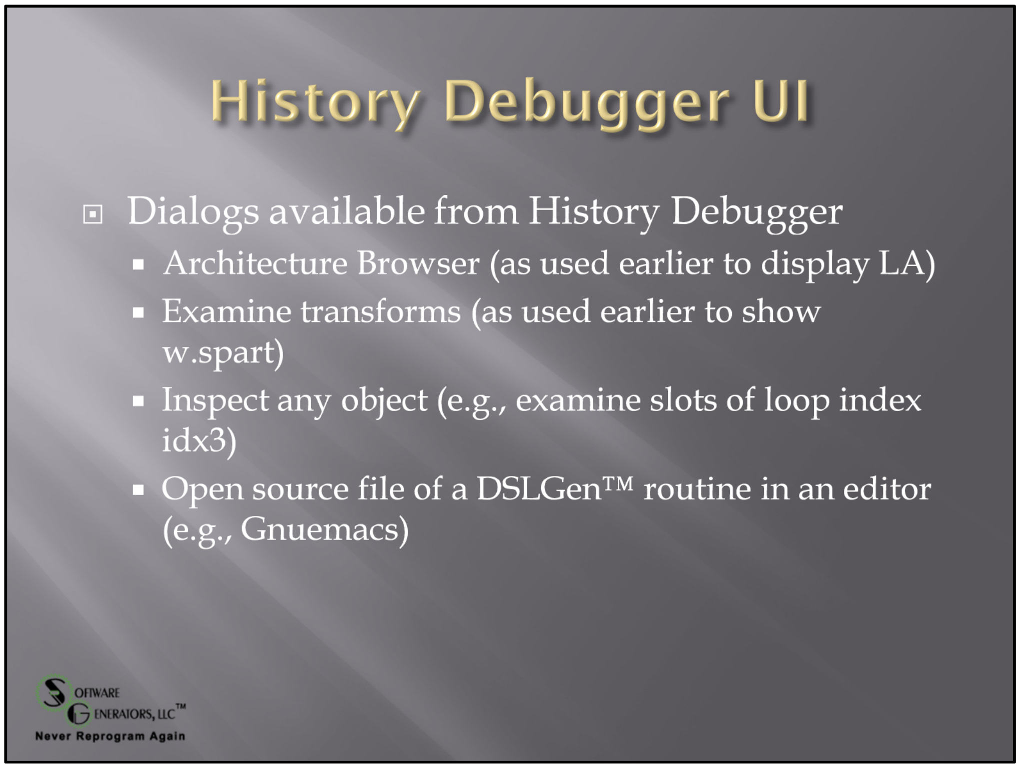

HISTORY-DEBUGGER
Tags... Pretty... Font... PrintCirc... Load History... Bookmarks... Search History...
Search History
RegExp Pattern
Search
From Beginning Case Sensitive Previous
Name Bindings Before After
Bindings Window
Bookmarks
Insert bookmark tagged ...
Move Up Move Down Delete
Outline Window
After Window
history-tree

```
((First loop2d APC generated) (partitioningcompositeleaf (5 5)))
((Second loop2d APC generated) (partitioningcompositeleaf (5 7)))
((Convolution Operator Processing) (partitioningconvolutiononleaves (5 10)))
((Create Edge1 Partition) (makeapartition (5 10 3 4)))
((Specialize Method-Transforms of Edge1) (specializeadsdefcomponent (5 10 3 4 4 10)))
((Merging Edge and Center APCs) (merge2dpixelloopsmethod-entry (5 49 18 2)))
((Loop2d2 + Loop2d3 -> Loop2d4) (generatedsdeclare (5 49 18 3)))
((Merging Edge1 and Edge6 forming Edge11) (makemergeddesignvbls (5 49 18 7)))
((Synch Design Decisions idx1 -> idx3) (sprefine23984 (6 5 2 1)))
((Refining Partestx to concrete form) (adtcomponentredirection (11 6)))
((Inlining Convolution Expression for Edge11) (dsoperatorredirection (11 14)))
((Inlining Subexpression -- Convolution Operator for Edge11) (partrightlinearproduct (11 14 1)))
((Inlining Subexpression -- W Method-Transform for Edge11) (w (11 14 4 2)))
```

```
(history-bookmarks (1))
(Generation Phase: createscopes (2))
(Generation Phase: declarations (3))
(Generation Phase: typeinfer (4))
(Generation Phase: localizeandpartition (5))
(Generation Phase: specrefine (6))
(Generation Phase: syntheticdesign (7))
(Generation Phase: partitioncodegen (8))
(Generation Phase: specrefine (9))
(Generation Phase: repartition (10))
(Generation Phase: formals (11))
(Generation Phase: simplifyloops (12))
(Generation Phase: insertscopes (13))
(runtime (14))
```

```
 scope2
 (body (assignop d
        (isqrtop (plusop (exptop (rightconvolutionop c spart) 2)
                         (exptop (rightconvolutionop c sppart) 2)))))))
```

SOFTWARE GENERATORS, LLC
Never Reprogram Again

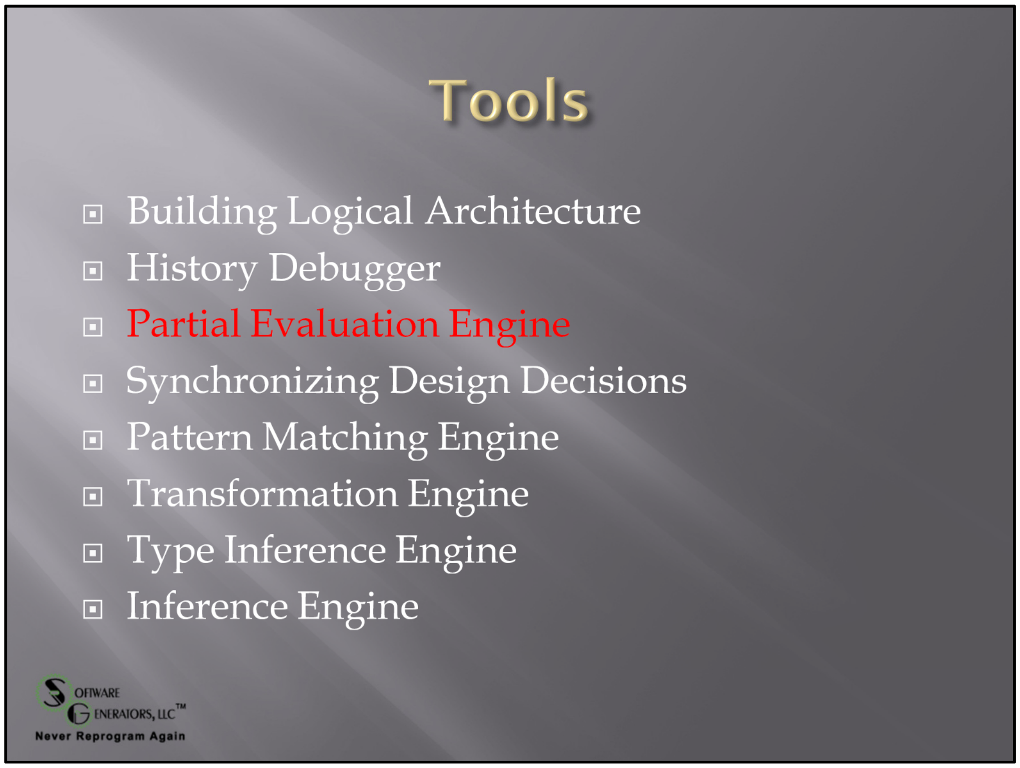


History Debugger UI

- Dialogs available from History Debugger
  - Architecture Browser (as used earlier to display LA)
  - Examine transforms (as used earlier to show w.spart)
  - Inspect any object (e.g., examine slots of loop index idx3)
  - Open source file of a DSLGen™ routine in an editor (e.g., Gnuemacs)

SOFTWARE GENERATORS, LLC™
Never Reprogram Again

Describe the user interface for the HD and if live demo is running, bring up one or more convenient dialogs such as the Architecture Browser, which has been illustrated earlier. Opening a DSLGen™ source file is probably too far into the weeds and would confuse more than inform.

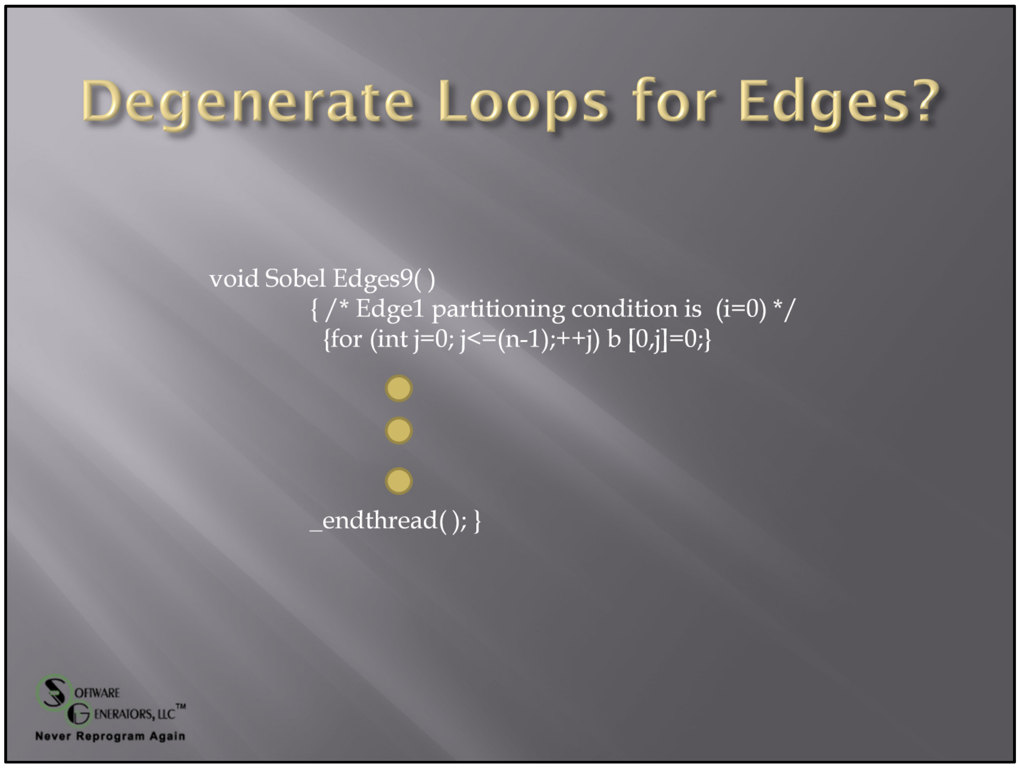


Tools

- Building Logical Architecture
- History Debugger
- Partial Evaluation Engine
- Synchronizing Design Decisions
- Pattern Matching Engine
- Transformation Engine
- Type Inference Engine
- Inference Engine

SOFTWARE GENERATORS, LLC™
Never Reprogram Again

Look at the partial evaluation engine in the context of our specific examples and, more specifically, see how the edge processing code gets inlined and simplified to what is seen in the early examples shown in the overview presentation.

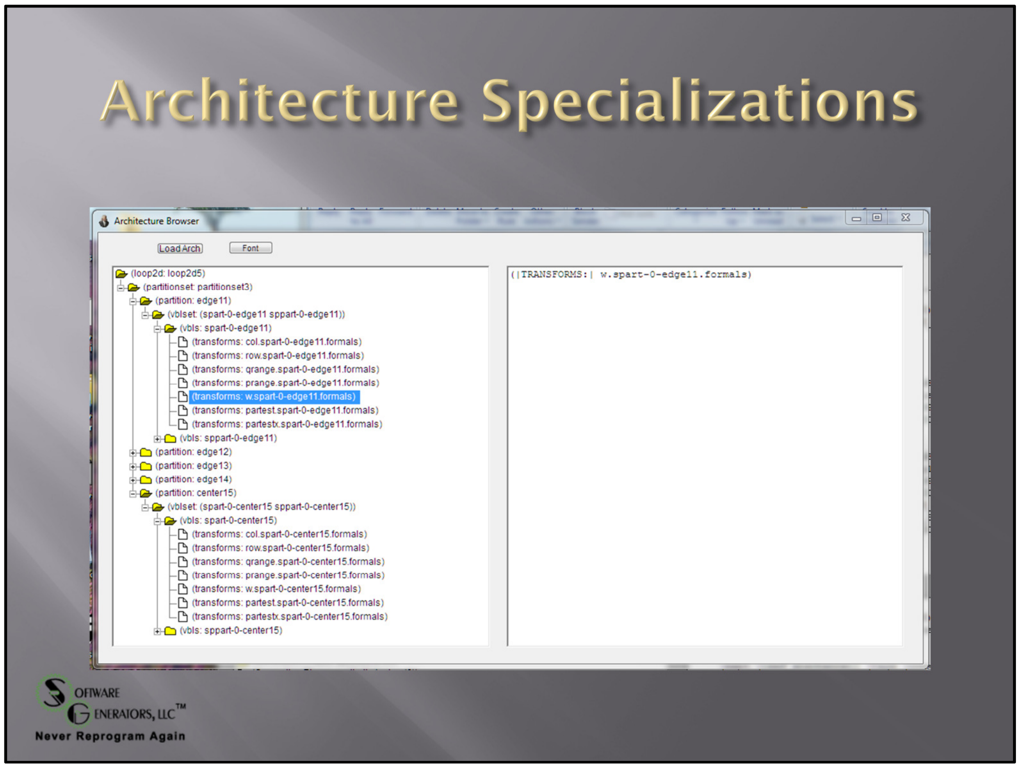


Degenerate Loops for Edges?

```
void Sobel Edges9( )
            { /* Edge1 partitioning condition is  (i=0) */
              {for (int j=0; j<=(n-1);++j) b [0,j]=0;}
                    ●
                    ●
                    ●
            _endthread( ); }
```

How did this code get derived?

We will derive the code for an edge example. The example will elide a some irrelevant complexities and do a bit of reordering of some steps to enhance the continuity of the overall process. Other than that, the example is quite faithful to the overall process.



Architecture Specializations

Architecture Browser

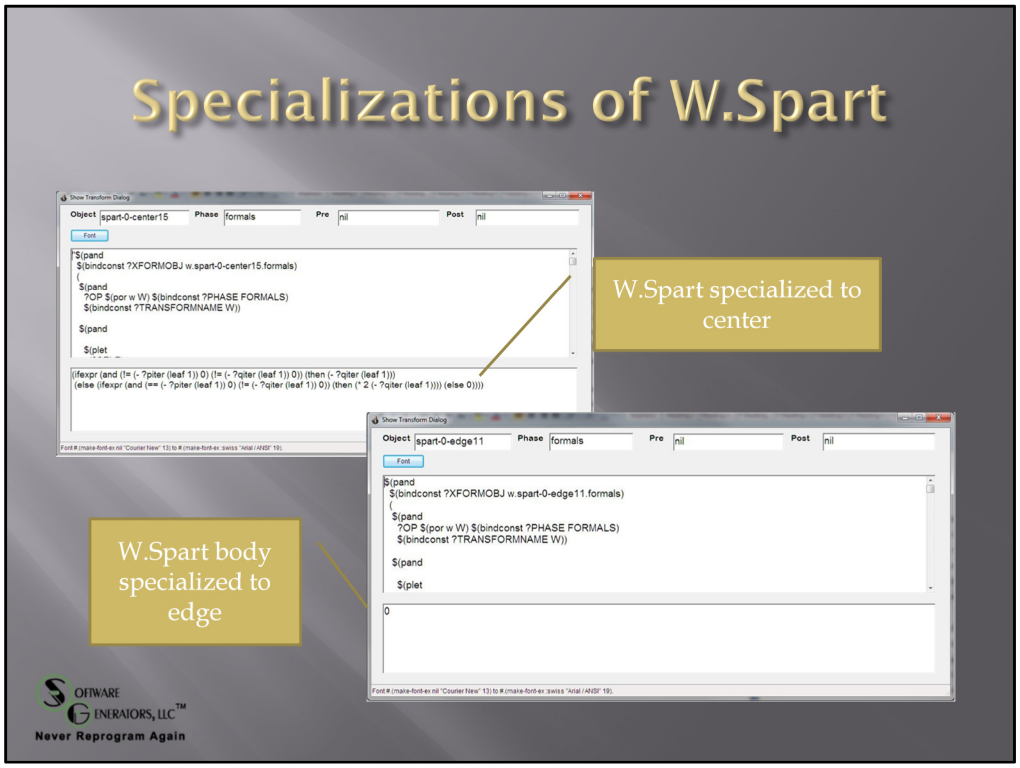
Load Arch Font

```
(loop2d: loop2d5)
  (partitionset: partitionset3)
    (partition: edge11)
      (vblset: (spart-0-edge11 sppart-0-edge11))
        (vbls: spart-0-edge11)
          (transforms: col.spart-0-edge11.formals)
          (transforms: row.spart-0-edge11.formals)
          (transforms: qrange.spart-0-edge11.formals)
          (transforms: prange.spart-0-edge11.formals)
          (transforms: w.spart-0-edge11.formals)
          (transforms: partest.spart-0-edge11.formals)
          (transforms: partestx.spart-0-edge11.formals)
        (vbls: sppart-0-edge11)
    (partition: edge12)
    (partition: edge13)
    (partition: edge14)
    (partition: center15)
      (vblset: (spart-0-center15 sppart-0-center15))
        (vbls: spart-0-center15)
          (transforms: col.spart-0-center15.formals)
          (transforms: row.spart-0-center15.formals)
          (transforms: qrange.spart-0-center15.formals)
          (transforms: prange.spart-0-center15.formals)
          (transforms: w.spart-0-center15.formals)
          (transforms: partest.spart-0-center15.formals)
          (transforms: partestx.spart-0-center15.formals)
        (vbls: sppart-0-center15)
```

```
(|TRANSFORMS:| w.spart-0-edge11.formals)
```

SOFTWARE GENERATORS, LLC™
Never Reprogram Again

Recall the definitions from the Logical Architecture seen in the earlier presentation. The following definition inlining and simplification steps will employ Method-Transforms (MT-s) that define w, row, col and other MT-s specialized for the two edge neighborhoods **spart-0-edge11** and **sppart-0-edge11**. All of this is in the context of a 2D loop constraint (specifically, **loop2D5**) that modifies a partially translated Problem Domain expression (e.g., **{b [i,j]= [(a[i,j] ⊕ spart-0-edge11[i,j])^2 + (a[i,j] ⊕ sppart-0-edge11[i,j])^2]$^{1/2}$}**).



Specializations of W.Spart

Show Transform Dialog

Object: spart-0-center15 Phase: formals Pre: nil Post: nil

```
'$(pand
 $(bindconst ?XFORMOBJ w.spart-0-center15.formals)
 (
  $(pand
   ?OP $(por w W) $(bindconst ?PHASE FORMALS)
   $(bindconst ?TRANSFORMNAME W))

  $(pand

   $(plet
```

```
(ifexpr (and (!= (- ?piter (leaf 1)) 0) (!= (- ?qiter (leaf 1)) 0)) (then (- ?qiter (leaf 1)))
 (else (ifexpr (and (== (- ?piter (leaf 1)) 0) (!= (- ?qiter (leaf 1)) 0)) (then (* 2 (- ?qiter (leaf 1)))) (else 0))))
```

Font #.(make-font-ex nil "Courier New" 13) to #.(make-font-ex :swiss "Arial / ANSI" 19).

W.Spart specialized to center

Show Transform Dialog

Object: spart-0-edge11 Phase: formals Pre: nil Post: nil

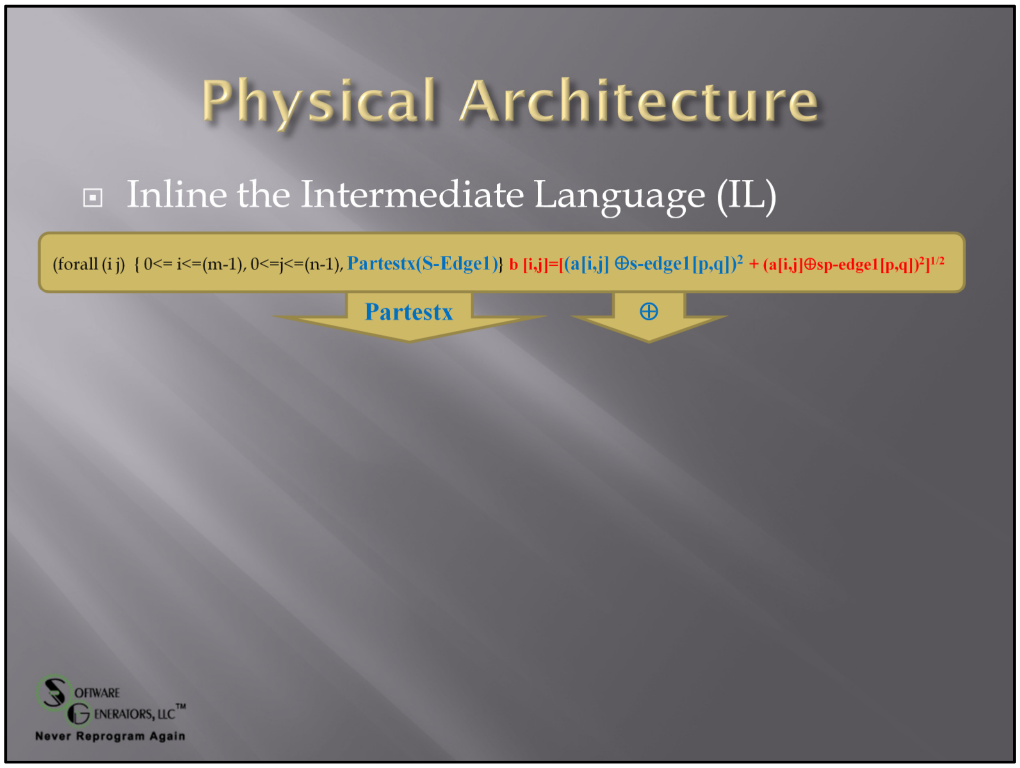
```
$(pand
 $(bindconst ?XFORMOBJ w.spart-0-edge11.formals)
 (
  $(pand
   ?OP $(por w W) $(bindconst ?PHASE FORMALS)
   $(bindconst ?TRANSFORMNAME W))

  $(pand

   $(plet
```

```
0
```

Font #.(make-font-ex nil "Courier New" 13) to #.(make-font-ex :swiss "Arial / ANSI" 19).

W.Spart body specialized to edge

Two definitions from the LA that we have seen earlier will have a profoundly different effect on the resulting code for center computations and edge computations. Specifically, recall the **MT definitions for w** (i.e., the Sobel weight function) of the center partition neighborhood and one of the edge partition neighborhoods. The RHS-s of these two definitions will produce differing resultant code for the respective partitions. The center neighborhood will produce the largely unsimplified code for the general case (with only the loop ranges modified to eliminate the edges). However, the edge neighborhood definition for w will always **result in 0**, **which will engender significant simplification of the edge pixel computations** (as we saw earlier in the example code). And it is this edge case simplification that we will follow through in the following slides.



Physical Architecture

- Inline the Intermediate Language (IL)

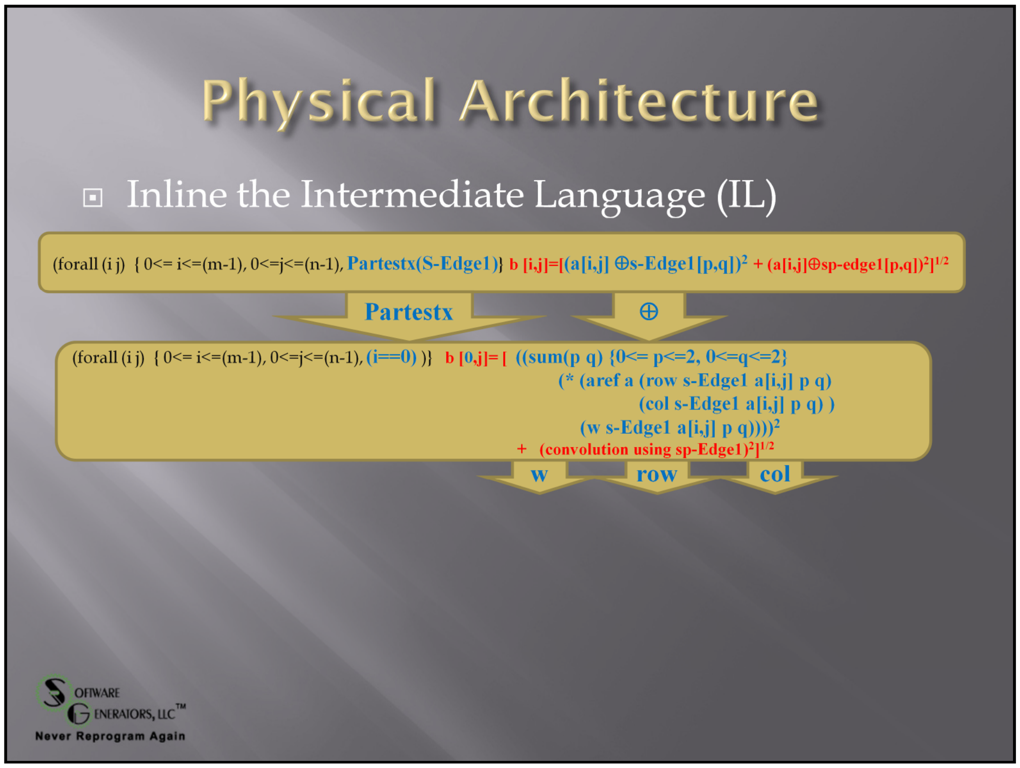
(forall (i j) { 0<= i<=(m-1), 0<=j<=(n-1), Partestx(S-Edge1)} b [i,j]=[(a[i,j] ⊕s-edge1[p,q])^2 + (a[i,j]⊕sp-edge1[p,q])^2]$^{1/2}$

Partestx
⊕
OFTWARE
ENERATORS, LLC™
Never Reprogram Again

So, let’s examine the refinement of one of the edge loops. We will follow this process through step by step and as we do, we will go to a concrete example of the actual step in the History Debugger. This will show the true forms on the AST, i.e., what a Domain Engineer debugging a derivation would see. Note, the concrete example we use is modestly more complex because of complexities in the real example that I have chosen to omit from the presentation example. Those complexities add nothing critical to the understanding of the process. The presentation example also omits dealing with the inner loop constraint over the neighborhood (i.e., the loop over p and q in the example). As the definitions are inlined in this example, the neighborhood loop (in the PL representation form) will pop into view at the appropriate point but the details of that constraint and its operation add no insights beyond those already provided by the operation of the loop constraint for the image (i.e., the loop over i and j). One additional proviso is that the real example (to be shown in the History Debugger) is dealing with different names (e.g., c rather than a) but the operational structure and transformations are obviously analogous.

We will follow through the refinement of the BLUE text portions of the loop expression. But the refinement of the convolution expression in RED text, while we ignore it for simplicity, is closely analogous.

So, as the first step, we need to inline the definition of the Partestx method transform (which refines to the specific partitioning condition for this loop) and the definition of the convolution operator.



Physical Architecture

- Inline the Intermediate Language (IL)

(forall (i j) { 0<= i<=(m-1), 0<=j<=(n-1), Partestx(S-Edge1)} b [i,j]=[(a[i,j] ⊕s-Edge1[p,q])^2 + (a[i,j]⊕sp-edge1[p,q])^2]$^{1/2}$

```
(forall (i j)  { 0<= i<=(m-1), 0<=j<=(n-1), (i==0) )}   b [0,j]= [  ((sum(p q) {0<= p<=2, 0<=q<=2}
                                                                     (* (aref a (row s-Edge1 a[i,j] p q)
                                                                                (col s-Edge1 a[i,j] p q) )
                                                                        (w s-Edge1 a[i,j] p q))))^2
                                                               +  (convolution using sp-Edge1)^2]^1/2
```

Partestx
⊕
w
row
col

SOFTWARE GENERATORS, LLC™
Never Reprogram Again

The Partestx MT refines pretty directly to the concrete partitioning condition "(i==0)". This step also rewrites the convolution definition in terms of IL abstractions of that definition.
The IL abstractions of note for convolution are the definition of w of the neighborhood (which we have seen earlier in the overview presentation) and the MT definitions of row and col, which map from neighborhood coordinates to image coordinates. This inlining of the convolution becomes the body of the neighborhood loop. The inlining of the neighborhood loop over p and q, though shown here to aid understanding of the context for the listener, is actually handled as a separate step.

The next slide will show the inlining of w, row and col but first, let's switch to a live instance of the History Debugger and see what the inlining steps look like in a real example.

(Reader's Note: If the presentation is not being presented live but just being read, the actions described below are simulated by the two following slides that show the History Debugger examining the two inline operations for the partitioning condition and the convolution. They are intended to provide a sense of what a live audience would see in the course of the following live demo steps.)

In the History Debugger using the image DSLGenICSRToolsDemoV4.dxl file, let's look at a concrete analogical example:

The analogical correspondences between the presentation example and the History Debugger example are:

a corresponds to c,
b corresponds to d,
i corresponds to idx3,
j corresponds to idx4,
s corresponds to spart.
sp corresponds to sppart.

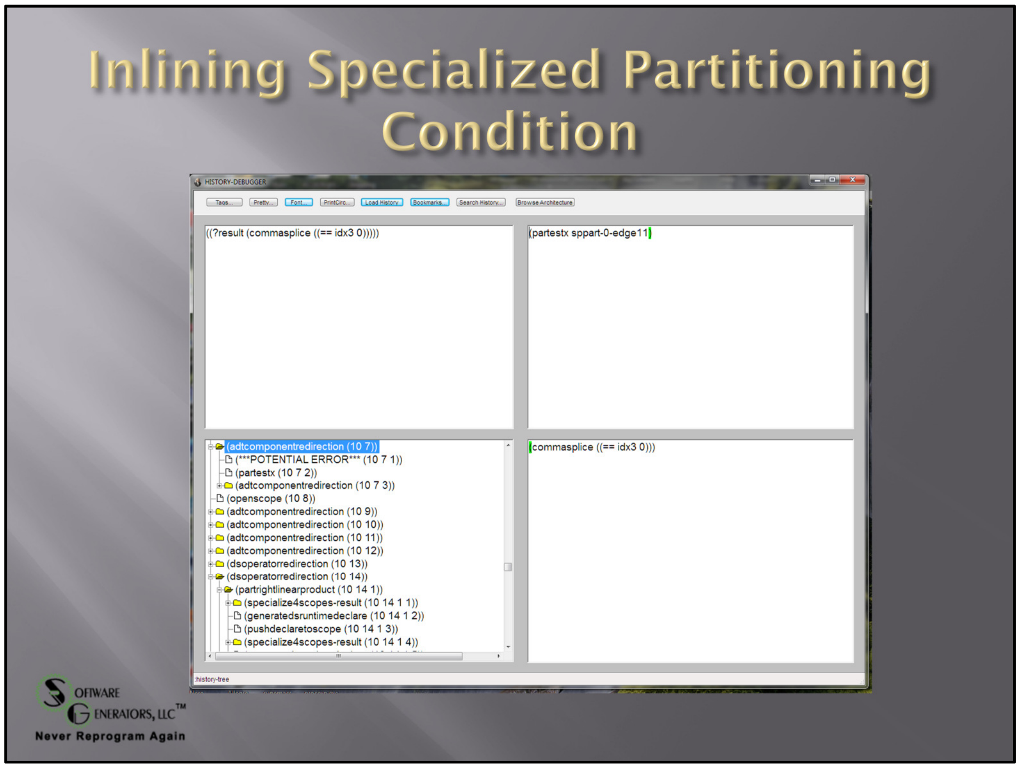
Note also: The real example is a RGB color image and therefore, introduces the added complexity of a loop over the RGB fields. Dsfieldop5 is the variable of that loop and it determines which color plane is being operated on. For the purposes of this example, we can just ignore the field operation because the rest is pretty much the same as a simpler grayscale image.

3. Physical Architecture: Inline Intermediate Language (Partestx ...)
   at **bookmark (((ParesttX results for neighborhood w.sppart-0-edge11)) (adtcomponentredirection (10 7)))**

a. Go to (adtcomponentredirection (10 7)) in the History Debugger
b. Show unrefined partitioning condition in the Before Window: (partestx sppart-0-edge11)
c. Show refined partitioning condition in the After Window: (commasplice ((== idx3 0)))

4. Physical Architecture: Inline Intermediate Language (rightconvolutionop ...)
   at bookmark **(((Conv Op for neighborhood spart-0-edge11)) (partrightlinearproduct (10 14 1)))**

```
a. Go to (partrightlinearproduct (10 14 1))
b. Show unrefined convolution operator in the Before Window:
                        (rightconvolutionop (dsfieldop5 (aref c idx3 idx4)) (aref sppart-0-edge11 P5 Q6))
c. Show refined convolution operator in the After Window:  (* (dsfieldop5 (aref c (row sppart-0-edge11 (aref c idx3 idx4) P5 Q6)
                                                                            (col sppart-0-edge11 (aref c idx3 idx4) P5 Q6)))
                                                              (w sppart-0-edge11 (aref c idx3 idx4) P5 Q6))
```



Inlining Specialized Partitioning Condition

```
((?result (commasplice ((== idx3 0)))))
```

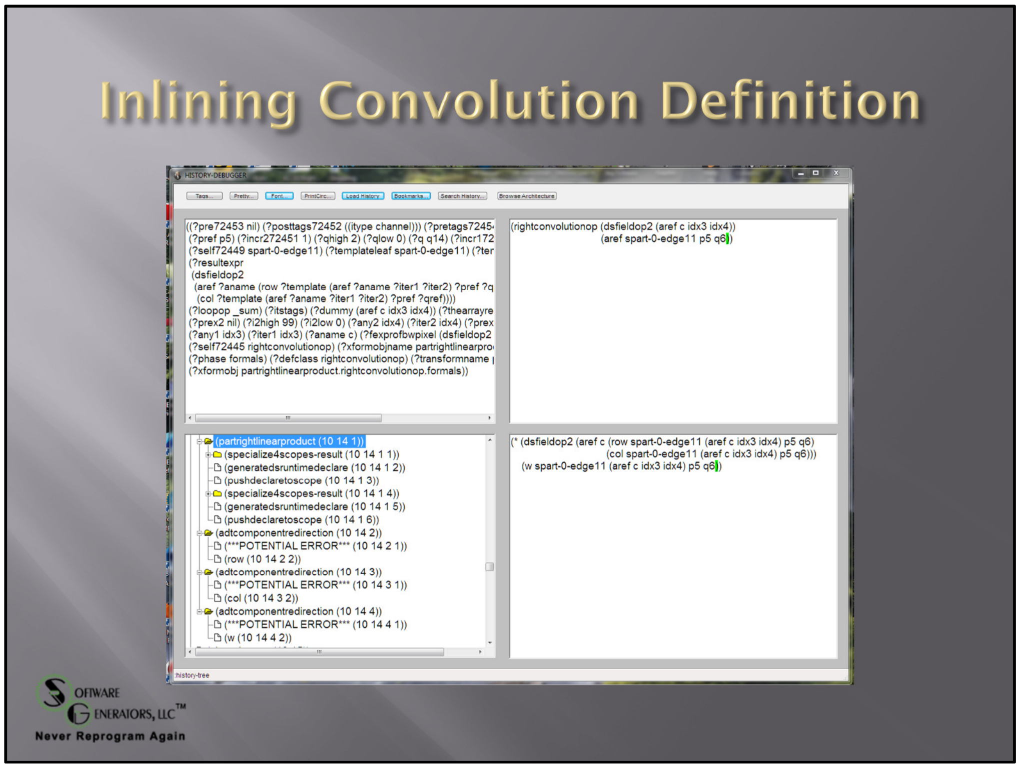
```
(partestx sppart-0-edge11)
```

```
(adtcomponentredirection (10 7))
  (***POTENTIAL ERROR*** (10 7 1))
  (partestx (10 7 2))
  (adtcomponentredirection (10 7 3))
(openscope (10 8))
(adtcomponentredirection (10 9))
(adtcomponentredirection (10 10))
(adtcomponentredirection (10 11))
(adtcomponentredirection (10 12))
(dsoperatorredirection (10 13))
(dsoperatorredirection (10 14))
  (partrightlinearproduct (10 14 1))
    (specialize4scopes-result (10 14 1 1))
    (generatedsruntimedeclare (10 14 1 2))
    (pushdeclaretoscope (10 14 1 3))
    (specialize4scopes-result (10 14 1 4))
```

```
(commasplice ((== idx3 0)))
```

OFTWARE
ENERATORS, LLC™
Never Reprogram Again

History Debugger showing the inlining of the partitioning condition MT during the "formals" phase of the generator. This **outline node** (i.e., **(adtcomponentredirection (10 7))**) is the summary of several sub-steps in the overall operation, the details of which are largely irrelevant to this level of discussion. Conceptually, this is a pretty straightforward inlining operation.



Inlining Convolution Definition

```
((?pre72453 nil) (?posttags72452 ((itype channel))) (?pretags7245
 (?pref p5) (?incr272451 1) (?qhigh 2) (?qlow 0) (?q q14) (?incr172
 (?self72449 spart-0-edge11) (?templateleaf spart-0-edge11) (?ter
 (?resultexpr
  (dsfieldop2
   (aref ?aname (row ?template (aref ?aname ?iter1 ?iter2) ?pref ?q
    (col ?template (aref ?aname ?iter1 ?iter2) ?pref ?qref))))
 (?loopop _sum) (?itstags) (?dummy (aref c idx3 idx4)) (?thearrayre
 (?prex2 nil) (?i2high 99) (?i2low 0) (?any2 idx4) (?iter2 idx4) (?prex
 (?any1 idx3) (?iter1 idx3) (?aname c) (?fexprofbwpixel (dsfieldop2
 (?self72445 rightconvolutionop) (?xformobjname partrightlinearpro
 (?phase formals) (?defclass rightconvolutionop) (?transformname
 (?xformobj partrightlinearproduct.rightconvolutionop.formals))
```

```
(rightconvolutionop (dsfieldop2 (aref c idx3 idx4))
                    (aref spart-0-edge11 p5 q6))
```

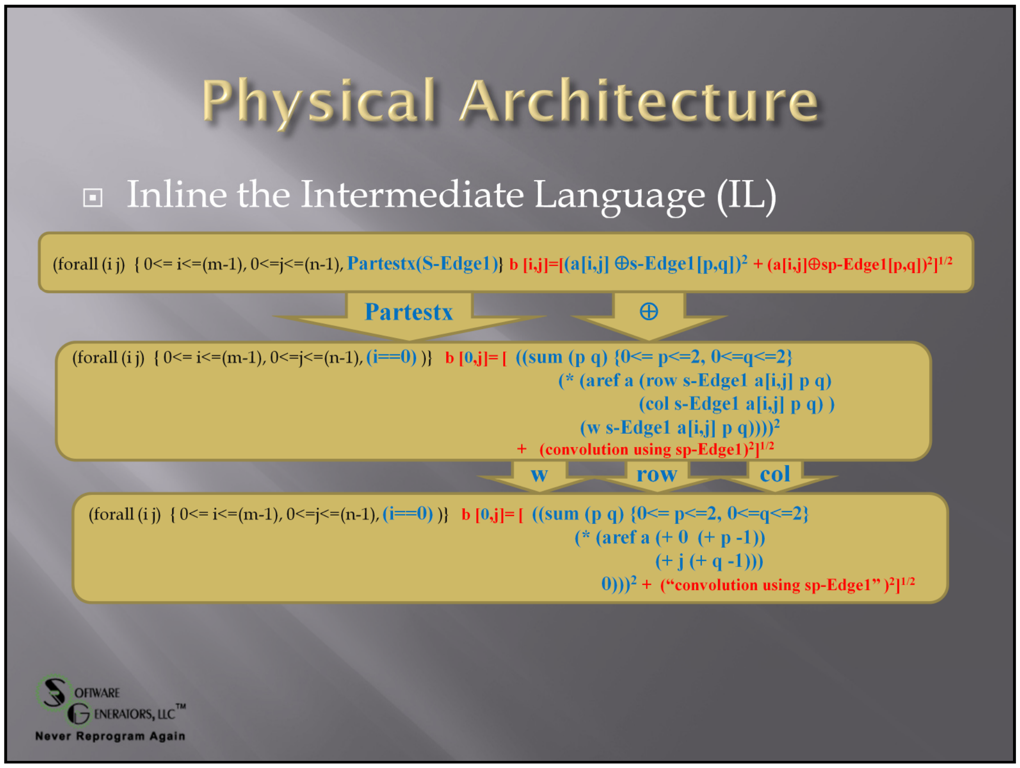
```
(partrightlinearproduct (10 14 1))
  (specialize4scopes-result (10 14 1 1))
  (generatedsruntimedeclare (10 14 1 2))
  (pushdeclaretoscope (10 14 1 3))
  (specialize4scopes-result (10 14 1 4))
  (generatedsruntimedeclare (10 14 1 5))
  (pushdeclaretoscope (10 14 1 6))
(adtcomponentredirection (10 14 2))
  (***POTENTIAL ERROR*** (10 14 2 1))
  (row (10 14 2 2))
(adtcomponentredirection (10 14 3))
  (***POTENTIAL ERROR*** (10 14 3 1))
  (col (10 14 3 2))
(adtcomponentredirection (10 14 4))
  (***POTENTIAL ERROR*** (10 14 4 1))
  (w (10 14 4 2))
```

```
(* (dsfieldop2 (aref c (row spart-0-edge11 (aref c idx3 idx4) p5 q6)
                       (col spart-0-edge11 (aref c idx3 idx4) p5 q6)))
   (w spart-0-edge11 (aref c idx3 idx4) p5 q6))
```

This slide is the History Debugger showing the step where the convolution definition (internally called "**rightconvolutionop**") is inlined during the "formals" generation phase. Notice that the history node **(partrightlinearproduct (10 14 1))** comprises a number of sub-steps that are doing a good deal of bookkeeping (e.g., creating declarations for newly generated C variable names and putting them into the correct scope). This level of detail is not relevant at this level of discussion and we shall ignore it. Fundamentally, this step is inlining the definition of the convolution operator, which is defined in terms of lower level intermediate language operators (i.e., lower level MT definitions). As we will see later in the example derivation, these lower level MT definitions will be drawn from that part of the Logical Architecture that is specialized for the edge11 partition and therefore, these edge11 specializations will cause the evolution of resultant code that is significantly different from the resultant code that would be produced for the center partition.

The details of the summation loop code structure generation is handled elsewhere in the derivation. Hence, the Domain Engineer doesn't see it at this step of the history tree. At this point, the Domain Engineer sees only the inlining of the convolution definition itself.

So, the next slide shows the recursive inlining of the lower level intermediate language definitions.



Physical Architecture

- Inline the Intermediate Language (IL)

```
(forall (i j)  { 0<= i<=(m-1), 0<=j<=(n-1), Partestx(S-Edge1)} b [i,j]=[(a[i,j] ⊕s-Edge1[p,q])^2 + (a[i,j]⊕sp-Edge1[p,q])^2]^1/2
```

Partestx ⊕

```
(forall (i j)  { 0<= i<=(m-1), 0<=j<=(n-1), (i==0) )}  b [0,j]= [  ((sum (p q) {0<= p<=2, 0<=q<=2}
                                                          (* (aref a (row s-Edge1 a[i,j] p q)
                                                                     (col s-Edge1 a[i,j] p q) )
                                                             (w s-Edge1 a[i,j] p q))))^2
                                                    +  (convolution using sp-Edge1)^2]^1/2
```

w row col

```
(forall (i j)  { 0<= i<=(m-1), 0<=j<=(n-1), (i==0) )}  b [0,j]= [  ((sum (p q) {0<= p<=2, 0<=q<=2}
                                                          (* (aref a (+ 0  (+ p -1))
                                                                     (+ j (+ q -1)))
                                                             0)))^2 +  (“convolution using sp-Edge1” )^2]^1/2
```

SOFTWARE GENERATORS, LLC™
Never Reprogram Again

Now, it is time to inline the definitions for <u>w</u>, <u>row</u> and <u>col</u>. These are conceptually pretty straightforward. Recall that the pixel upon which the neighborhood is centered will be at the image position a[0,j]. <u>Row</u> maps from neighborhood coordinates (i.e., “p”) to the coordinates of the image pixel corresponding to that neighborhood position (i.e., “(+ 0 (+ p -1))” where the image row coordinate “0” arises because this is an edge pixel). <u>Col</u> does an analogous mapping for the column coordinates. Thus, the target image pixel will be at a[(+ 0 (+ p -1)) , (+ j (+q -1))]. And because the pixel in this example is an edge pixel, the (w ….) expression will inline to 0.

(<u>Nitpicking detail in the weeds</u>: The partitioning condition “(i==0)” will constrain the range of p to be [1,2] (i.e., 1<=p<=2), here expressed using the C rather than Image Algebra indexing system (i.e., [0,2] rather than [-1,1] indexing system). That is, if p could be 0, then the index “i" would be -1 and would fall outside of the actual image. This constraint is irrelevant for this example because w is everywhere 0 and thus, the summation will be 0 regardless. In a problem where w is non-zero (e.g., image averaging), the ranges of the summation loops will vary and correctly reflect this constraint. How this happens is too far in the weeds for this example. However, the short answer is that domain knowledge (e.g., “edge-ness”) triggers the injection of specialized definitions for <u>row</u> and <u>col</u> that incorporate the proper constraints. But because these details are largely irrelevant for this example, I have omitted them. This note is here in case someone catches the problem implied by the simplification.)

Now, let’s switch to the History Debugger (HD) and see an actual example of these inlining steps. **(Reader’s note: Even though there are no actual screen shots of the History Debugger for these cases, the Before and After Window values are shown in the notes below. And though you do not get to see the actual Bindings and Outline windows values of the HD, they are not highly relevant to this level of discussion. It should be clear what it going on here from the notes below.)**

5. Physical Architecture: Inline Intermediate Language (w ...)
   at bookmark **(((W Method-Transform for neighborhood spart-0-edge11)) (adtcomponentredirection (10 14 4)))**

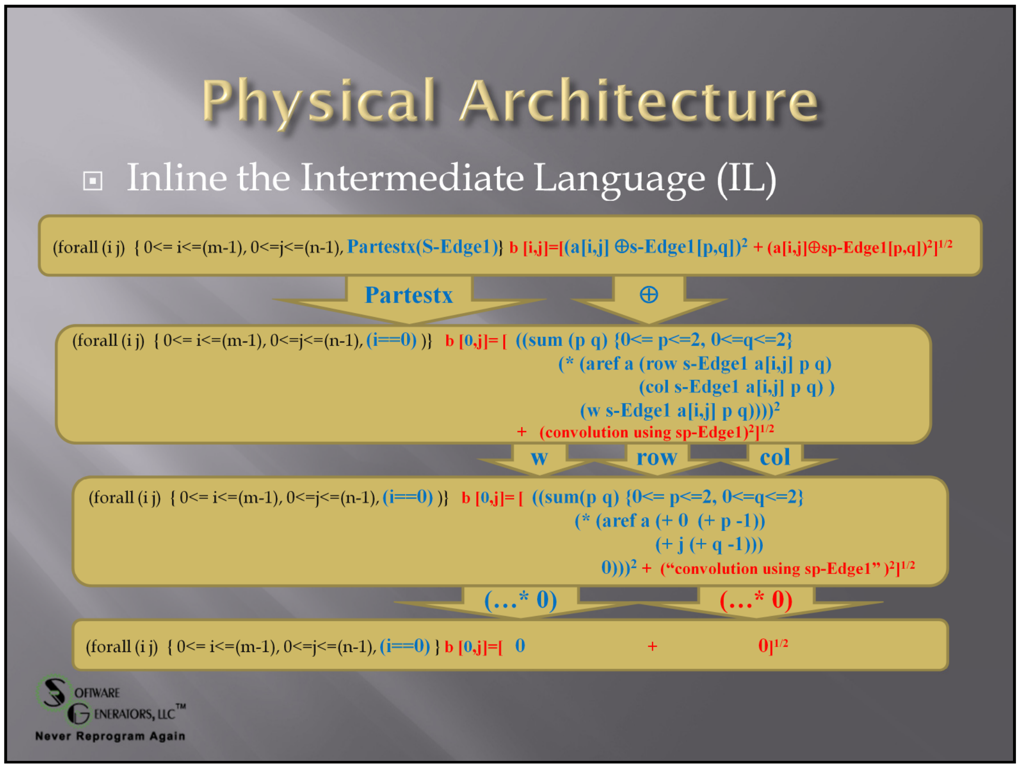
   a. Go to (adtcomponentredirection (10 14 4)) in the History Debugger
   b. Show unrefined intermediate language expression of w in the Before Window:
      (w sppart-0-edge11 (aref c idx3 idx4) P5 Q6)
   c. Show refined intermediate language expression w in the After Window: 0

6. Physical Architecture: Inline Intermediate Language (row ...)
   at bookmark **(((Row Method-Transform for neighborhood spart-0-edge11)) (adtcomponentredirection (10 14 2)))**

   a. Go to (adtcomponentredirection (10 14 2)) in the History Debugger
   b. Show unrefined intermediate language expression of row in the Before Window:
      (row sppart-0-edge11 (aref c idx3 idx4) P5 Q6)
   c. Show refined intermediate language expression row in the After Window: (+ idx3 (+ P5 -1))

7. Physical Architecture: Inline Intermediate Language (col ...)
   at bookmark **(((Col Method-Transform for neighborhood spart-0-edge11)) (adtcomponentredirection (10 14 3)))**

   a. Go to (adtcomponentredirection (10 14 3)) in the History Debugger
   b. Show unrefined intermediate language expression of col in the Before Window:
      (col sppart-0-edge11 (aref c idx3 idx4) P5 Q6)
   c. Show refined intermediate language expression col in the After Window: (+ idx4 (+ Q6 -1))



Physical Architecture

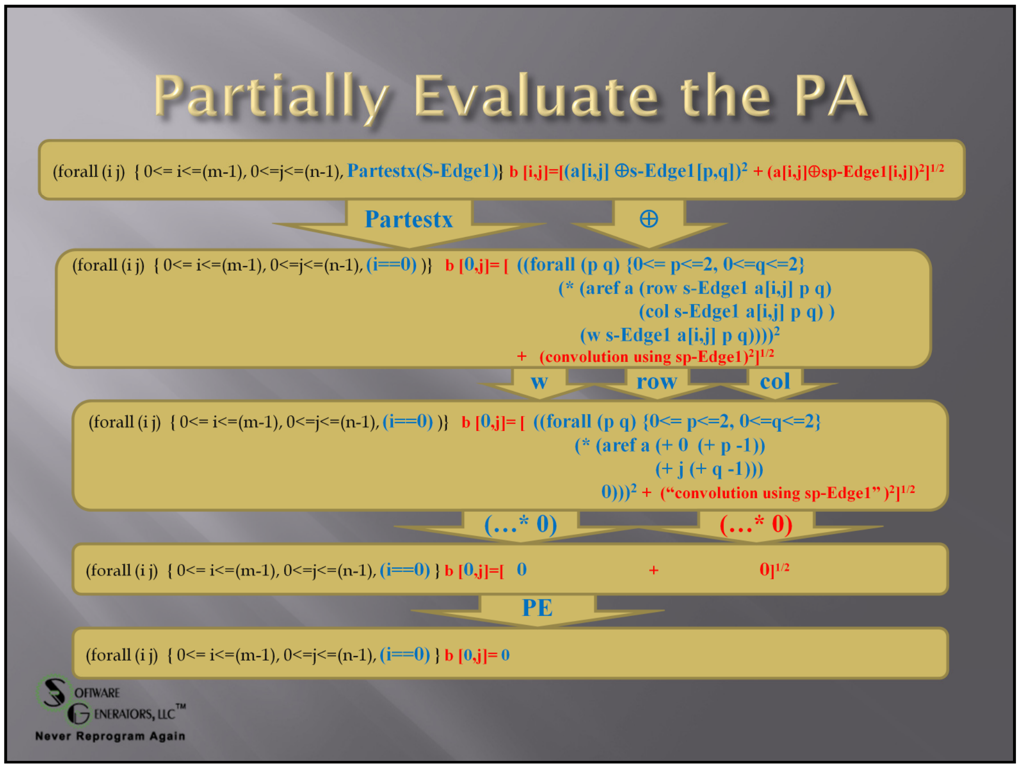
- Inline the Intermediate Language (IL)

```
(forall (i j)  { 0<= i<=(m-1), 0<=j<=(n-1), Partestx(S-Edge1)} b [i,j]=[(a[i,j] ⊕s-Edge1[p,q])^2 + (a[i,j]⊕sp-Edge1[p,q])^2]^1/2

(forall (i j)  { 0<= i<=(m-1), 0<=j<=(n-1), (i==0) )}   b [0,j]= [  ((sum (p q) {0<= p<=2, 0<=q<=2}
                                                                        (* (aref a (row s-Edge1  a[i,j]  p q)
                                                                                   (col s-Edge1  a[i,j]  p q) )
                                                                           (w s-Edge1  a[i,j]  p q))))^2
                                                                   +  (convolution using sp-Edge1)^2]^1/2

(forall (i j)  { 0<= i<=(m-1), 0<=j<=(n-1), (i==0) )}   b [0,j]= [  ((sum(p q) {0<= p<=2, 0<=q<=2}
                                                                        (* (aref a (+ 0  (+ p -1))
                                                                                   (+ j (+ q -1)))
                                                                           0)))^2 +  ("convolution using sp-Edge1" )^2]^1/2

(forall (i j)  { 0<= i<=(m-1), 0<=j<=(n-1), (i==0) } b [0,j]=[   0                +                0]^1/2
```

To show simplification of i to 0 and the pixel value expression (* … 0) to 0, go to "simplifyloops" phase in the History Debugger. **(Reader's note: The partial evaluation of this expression is so blindingly obvious that not actually seeing the step in the HD is not a great sacrifice.)**

8. Physical Architecture: Simplifyforallloops-pe (pe ...)
at bookmark **((Partially evaluate Conv Expr for neighborhood spart-0-edge11) (pe (11 7 1 2 3 3)))**

a. Go to step (pe (11 7 1 2 3 3)) in the History Debugger
b. Show un-partially evaluated expression in the Before Window:
   (* (dsfieldop5 (aref c (+ idx3 (+ P5 -1)) (+ idx4 (+ Q6 -1)))) 0)
c. Show partially evaluated expression in the After Window: 0



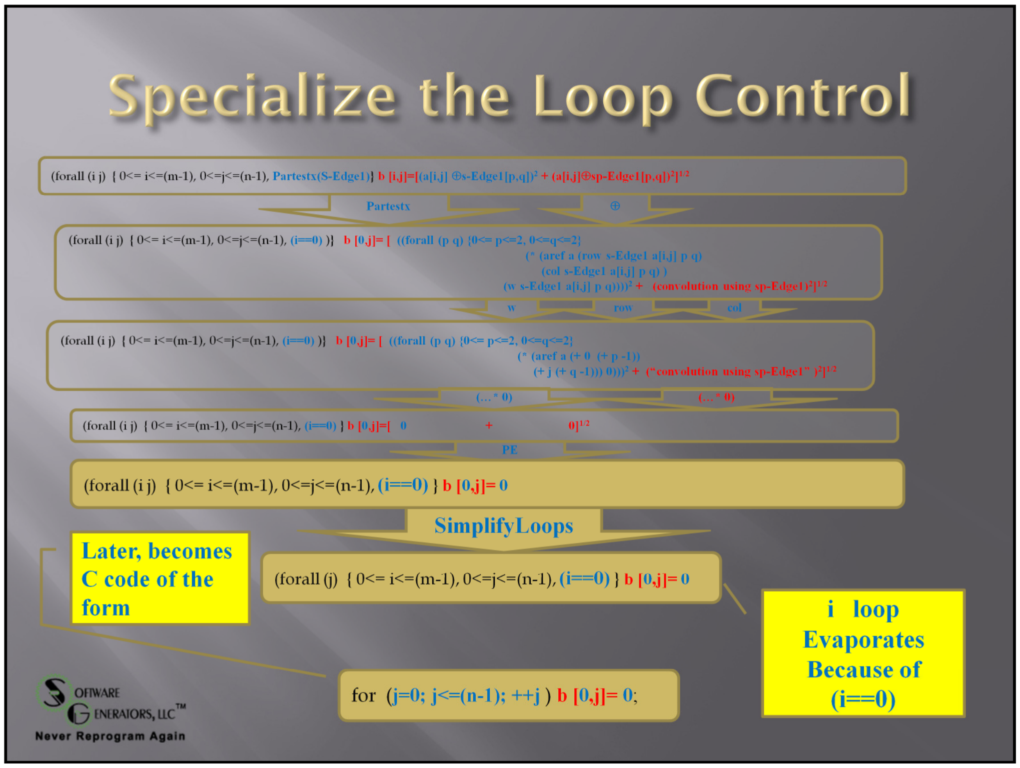
Partially Evaluate the PA

```
(forall (i j)  { 0<= i<=(m-1), 0<=j<=(n-1), Partestx(S-Edge1)} b [i,j]=[(a[i,j] ⊕s-Edge1[p,q])^2 + (a[i,j]⊕sp-Edge1[i,j])^2]^1/2
        Partestx          ⊕
(forall (i j)  { 0<= i<=(m-1), 0<=j<=(n-1), (i==0) )}  b [0,j]= [  ((forall (p q) {0<= p<=2, 0<=q<=2}
                                                                      (* (aref a (row s-Edge1 a[i,j] p q)
                                                                                 (col s-Edge1 a[i,j] p q) )
                                                                         (w s-Edge1 a[i,j] p q))))^2
                                                                 +  (convolution using sp-Edge1)^2]^1/2
        w     row     col
(forall (i j)  { 0<= i<=(m-1), 0<=j<=(n-1), (i==0) )}  b [0,j]= [  ((forall (p q) {0<= p<=2, 0<=q<=2}
                                                                      (* (aref a (+ 0  (+ p -1))
                                                                                 (+ j (+ q -1)))
                                                                         0)))^2 +  (“convolution using sp-Edge1” )^2]^1/2
        (…* 0)        (…* 0)
(forall (i j)  { 0<= i<=(m-1), 0<=j<=(n-1), (i==0) } b [0,j]=[  0          +          0]^1/2
        PE
(forall (i j)  { 0<= i<=(m-1), 0<=j<=(n-1), (i==0) } b [0,j]= 0
```

To show simplification of (0 + 0) to 0, go to simplifyallloops phase. **(Reader's note: Again, this is blindingly obvious. The actual mechanics of this simplification in a real example are a bit less straightforward than presented here but while the variation may be mildly amusing, it provides no deep insights into the generation process. So, again. Beyond the empirical evidence of an actual implementation, nothing is lost by not seeing the HD artifacts.)**

9. Physical Architecture: The step that partially evaluates the expression – (isqrt 0^{**2} + 0^{**2}), which is the RHS of an assignment (:=) statement,
 at bookmark **((Square Root of (0 + 0), RHS of edge11 loop) (partitioningsimplifyletbodies (11 8)))**

a. Go to step (partitioningsimplifyletbodies (11 8))
i. (pe (11 8 1 2 2 3)) - PE of (0 + 0)
ii. (pe (11 8 1 2 3 2 2)) - PE of (isqrt (+ (expr 0 2) (expr 0 2))
b. Un-partially evaluated expr in assigment statement in the Before Window:
.... (:= (dsfieldop2 (aref d idx3 idx4)) (isqrt 0^{**2} + 0^{**2}))
c. Partially evaluated result in assigment statementin the After Window:
.... (:= (dsfieldop2 (aref d idx3 idx4)) 0)



Specialize the Loop Control

```
(forall (i j)  { 0<= i<=(m-1), 0<=j<=(n-1), Partestx(S-Edge1)} b [i,j]=[(a[i,j] ⊕s-Edge1[p,q])² + (a[i,j]⊕sp-Edge1[p,q])²]^1/2
```

```
(forall (i j)  { 0<= i<=(m-1), 0<=j<=(n-1), (i==0) )}   b [0,j]= [  ((forall (p q) {0<= p<=2, 0<=q<=2}
                                                                        (* (aref a (row s-Edge1 a[i,j] p q)
                                                                           (col s-Edge1 a[i,j] p q) )
                                                                      (w s-Edge1 a[i,j] p q))))² +  (convolution using sp-Edge1)²]^1/2
```

```
(forall (i j)  { 0<= i<=(m-1), 0<=j<=(n-1), (i==0) )}   b [0,j]= [  ((forall (p q) {0<= p<=2, 0<=q<=2}
                                                                        (* (aref a (+ 0  (+ p -1))
                                                                           (+ j (+ q -1))) 0)))² +  ("convolution using sp-Edge1" )²]^1/2
```

```
(forall (i j)  { 0<= i<=(m-1), 0<=j<=(n-1), (i==0) } b [0,j]=[   0              +              0]^1/2
```

```
(forall (i j)  { 0<= i<=(m-1), 0<=j<=(n-1), (i==0) } b [0,j]= 0
```

```
(forall (j)  { 0<= i<=(m-1), 0<=j<=(n-1), (i==0) } b [0,j]= 0
```

Later, becomes C code of the form

i loop Evaporates Because of (i==0)

```
for  (j=0; j<=(n-1); ++j ) b [0,j]= 0;
```

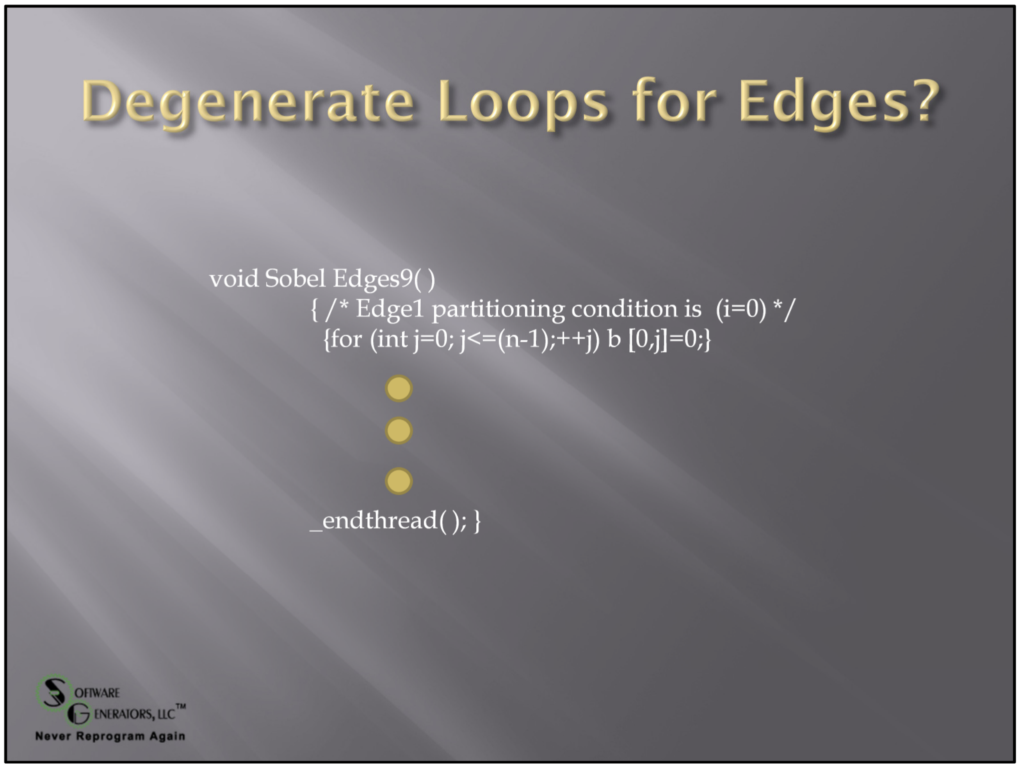
In the generator phase that performs most of the **loop simplification** in this example (which appropriately enough is called the "SimplifyLoops" phase), the loop form is reduced from a loop over "i and j" to a loop over just "j". One assertion in the partitioning condition constraint list (i.e., the "(i==0)" assertion) causes the loop over "i" to evaporate and all instances of "i" in the body to be replaced with "0". (Recall that this concrete form "(i==0)" was refined earlier from the generic Intermediate Language form "Partestx(S-Edge1)".)

We can view this step in an actual example by switching to a running instance of the History Debugger (if an instance is running during this presentation).

Goto: **(partitioningsimplifyforallloops (11 11))** for full transformation from **(forall (idx3 idx4) ….)** to **(forall (idx4) ….)** when the PC is "**(idx3 == 0)**",
at bookmark **(((Loop over Idx3 evaporates when (Idx3 == 0)) (partitioningsimplifyforallloops (11 11))**

a. Go to history node (partitioningsimplifyforallloops (11 11))
b. Generic loop for RGB assignment statements in the Before window:
   (_forall (idx3 idx4) …RGB assignments here …)
c. Refined loop spec for RGB assignment statements in the After window:
   (_forall (idx4) …RGB assignments here …)

Click for next build of the slide:
To make it clear where this process will eventually end up, this slide shows a C-like textual form for the eventual result. **But to be clear**, the **textual** form of the **C code is produced much later** in the generation process and long after the loop evaporation step. Text based code generation, which adds the surface C syntax to the AST, is the very last phase in the overall generation process.



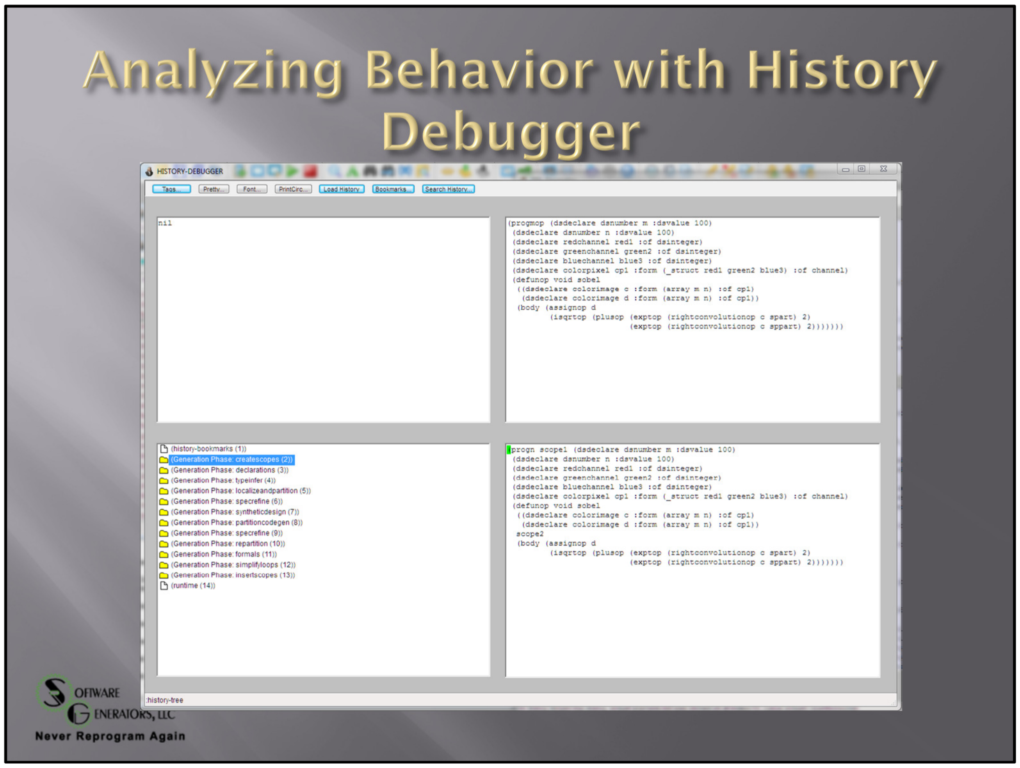
Degenerate Loops for Edges?

```
void Sobel Edges9( )
        { /* Edge1 partitioning condition is  (i=0) */
          {for (int j=0; j<=(n-1);++j) b [0,j]=0;}

              ●
              ●

              ●
        _endthread( ); }
```

OFTWARE
ENERATORS, LLC™
Never Reprogram Again

The overall simplification process results in the form that we saw earlier in the example C code.

Analyzing Behavior with History Debugger



```
nil
```

```
(progmop (dsdeclare dsnumber m :dsvalue 100)
 (dsdeclare dsnumber n :dsvalue 100)
 (dsdeclare redchannel red1 :of dsinteger)
 (dsdeclare greenchannel green2 :of dsinteger)
 (dsdeclare bluechannel blue3 :of dsinteger)
 (dsdeclare colorpixel cp1 :form (_struct red1 green2 blue3) :of channel)
 (defunop void sobel
  ((dsdeclare colorimage c :form (array m n) :of cp1)
   (dsdeclare colorimage d :form (array m n) :of cp1))
  (body (assignop d
         (isqrtop (plusop (exptop (rightconvolutionop c spart) 2)
                          (exptop (rightconvolutionop c sppart) 2)))))))
```

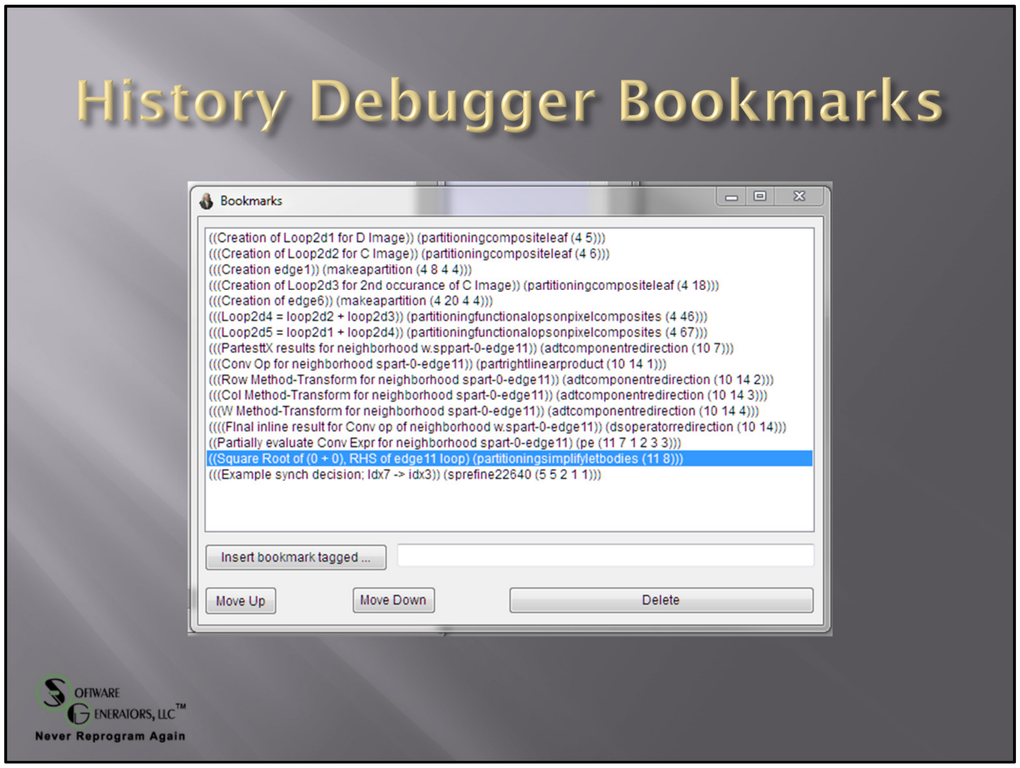
```
(history-bookmarks (1))
(Generation Phase: createscopes (2))
(Generation Phase: declarations (3))
(Generation Phase: typeinfer (4))
(Generation Phase: localizeandpartition (5))
(Generation Phase: specrefine (6))
(Generation Phase: syntheticdesign (7))
(Generation Phase: partitioncodegen (8))
(Generation Phase: specrefine (9))
(Generation Phase: repartition (10))
(Generation Phase: formals (11))
(Generation Phase: simplifyloops (12))
(Generation Phase: insertscopes (13))
(runtime (14))
```

```
(progn scope1 (dsdeclare dsnumber m :dsvalue 100)
 (dsdeclare dsnumber n :dsvalue 100)
 (dsdeclare redchannel red1 :of dsinteger)
 (dsdeclare greenchannel green2 :of dsinteger)
 (dsdeclare bluechannel blue3 :of dsinteger)
 (dsdeclare colorpixel cp1 :form (_struct red1 green2 blue3) :of channel)
 (defunop void sobel
  ((dsdeclare colorimage c :form (array m n) :of cp1)
   (dsdeclare colorimage d :form (array m n) :of cp1))
  scope2
  (body (assignop d
         (isqrtop (plusop (exptop (rightconvolutionop c spart) 2)
                          (exptop (rightconvolutionop c sppart) 2)))))))
```

(Optionally, you may want to skip this slide and the next one.)

We have all ready covered much of the introductory ground for the History Debugger in previous slides. This slide is here largely to provide a context for the use of the bookmark dialog in the next slide, which gets to the partial evaluation step that reduces the square root expression to zero for the edge partitions.

In a live demo, it may be of some interest to select the **runtime node** and show the summary of the overall generation process. It shows the computation time (typically for this example, around **75 seconds**), the size of the history tree (typically around **2700-2800 nodes**, depending on how many extra traces are turned on) and the number of true transformations that trigger (typically for this example, around **800 transformations**).

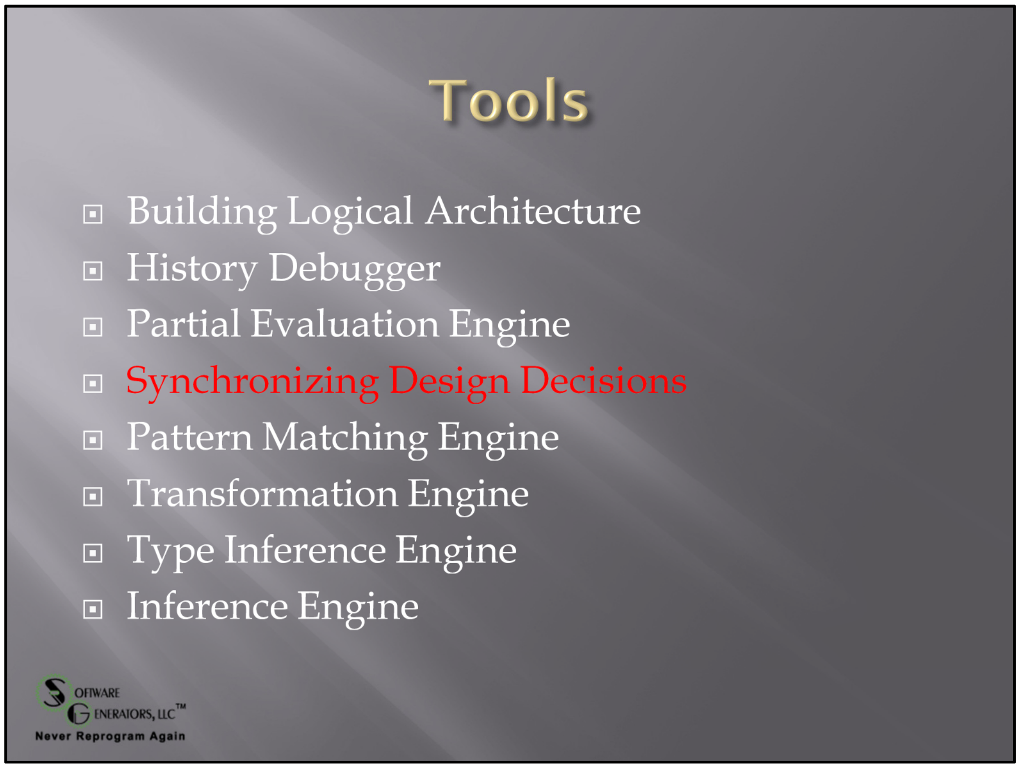


History Debugger Bookmarks

```
((Creation of Loop2d1 for D Image)) (partitioningcompositeleaf (4 5)))
(((Creation of Loop2d2 for C Image)) (partitioningcompositeleaf (4 6)))
(((Creation edge1)) (makeapartition (4 8 4 4)))
(((Creation of Loop2d3 for 2nd occurance of C Image)) (partitioningcompositeleaf (4 18)))
(((Creation of edge6)) (makeapartition (4 20 4 4)))
(((Loop2d4 = loop2d2 + loop2d3)) (partitioningfunctionalopsonpixelcomposites (4 46)))
(((Loop2d5 = loop2d1 + loop2d4)) (partitioningfunctionalopsonpixelcomposites (4 67)))
(((PartesttX results for neighborhood w.sppart-0-edge11)) (adtcomponentredirection (10 7)))
(((Conv Op for neighborhood spart-0-edge11)) (partrightlinearproduct (10 14 1)))
(((Row Method-Transform for neighborhood spart-0-edge11)) (adtcomponentredirection (10 14 2)))
(((Col Method-Transform for neighborhood spart-0-edge11)) (adtcomponentredirection (10 14 3)))
(((W Method-Transform for neighborhood spart-0-edge11)) (adtcomponentredirection (10 14 4)))
((((FInal inline result for Conv op of neighborhood w.spart-0-edge11)) (dsoperatorredirection (10 14)))
((Partially evaluate Conv Expr for neighborhood spart-0-edge11) (pe (11 7 1 2 3 3)))
((Square Root of (0 + 0), RHS of edge11 loop) (partitioningsimplifyletbodies (11 8)))
(((Example synch decision; Idx7 -> idx3)) (sprefine22640 (5 5 2 1 1)))
```

History Debugger bookmarks used in the previous slides deriving the Physical architecture via inlining and Partial Evaluation.

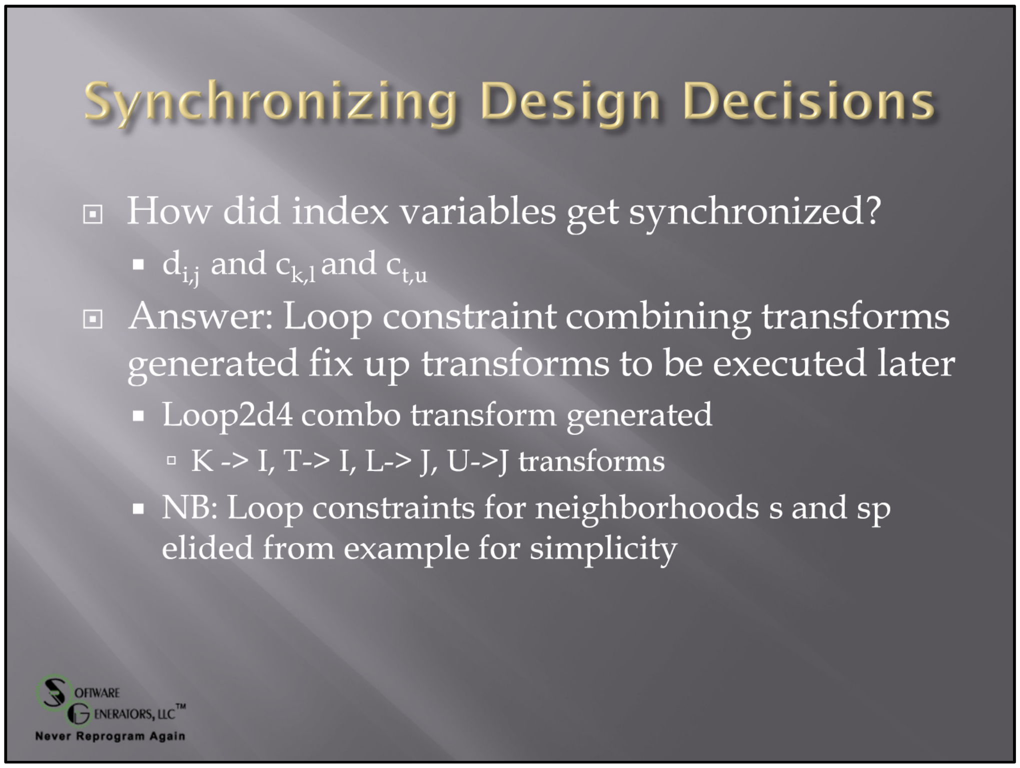
(Reader's note: This slide is here largely to provide a link to the point in the history where the fully inlined expression is handed to the partial evaluator to see if it can be simplified, which in the case of the edge computations, it can. In a presentation in which a live version of the DSLGen™ is running, the speaker has the option of using this bookmark to go directly to the point in the history tree where this simplification happens for the edge11 context. It should be pretty obvious that the inlined arithmetic expression in the previous slide will simplify to zero.)



Tools

- Building Logical Architecture
- History Debugger
- Partial Evaluation Engine
- Synchronizing Design Decisions
- Pattern Matching Engine
- Transformation Engine
- Type Inference Engine
- Inference Engine

SOFTWARE GENERATORS, LLC™
Never Reprogram Again



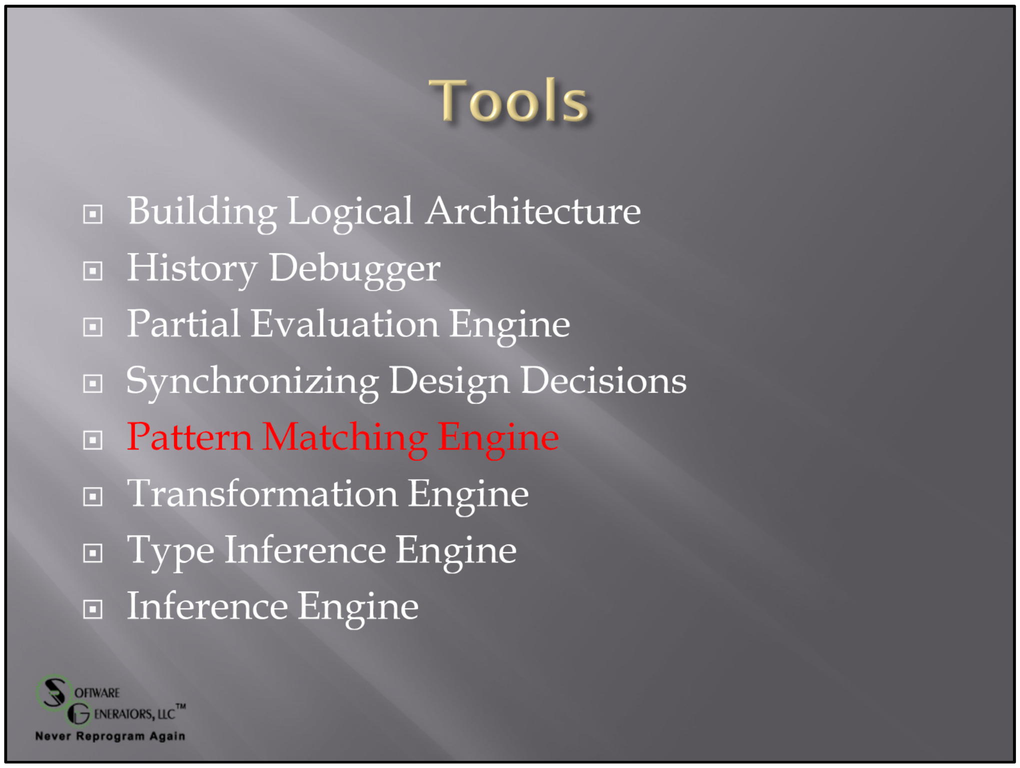
Synchronizing Design Decisions

- How did index variables get synchronized?
  - $d_{i,j}$ and $c_{k,l}$ and $c_{t,u}$
- Answer: Loop constraint combining transforms generated fix up transforms to be executed later
  - Loop2d4 combo transform generated
    - K -> I, T-> I, L-> J, U->J transforms
  - NB: Loop constraints for neighborhoods s and sp elided from example for simplicity

OFTWARE
ENERATORS, LLC™
Never Reprogram Again

Recall the earlier build/animation slide that illustrated translation of domain oriented expressions on images into domain oriented expressions on pixels. Lots of things got created during this process: names of image loop(s) indexes, names of neighborhood loop(s) indexes, loop constraints, partition constraints, etc. Recall that several different pairs of index names were created to process image b, image a, and the two convolution expressions of image a. As the process proceeded, it was step by step determined that only one set (of indexes) was needed and the other indexes were abandoned. Unfortunately, the abandon index names were sprinkled over the domain expression and elsewhere. So, parts of the AST were out of synch with other parts of the AST after that phase of translation finished. How does this get resolved by the generator?

Answer: Whenever a set (of indexes) is abandoned, a transformation is dynamically generated that is enabled to fire during a later synchronization phase (the“specrefine” phase). When the logical architecture building phase is complete, DSLGen™ walks over the AST during a specrefine phase allowing these “fix up” transformations to synchronize the overall design. **(Speaker note: if a live instance of DSLGen™ is running, there is a bookmark for one of the fix ups. Although, the idea is intuitive enough that it is probably a waste of time unless the audience really wants to see under the “hood”.)**

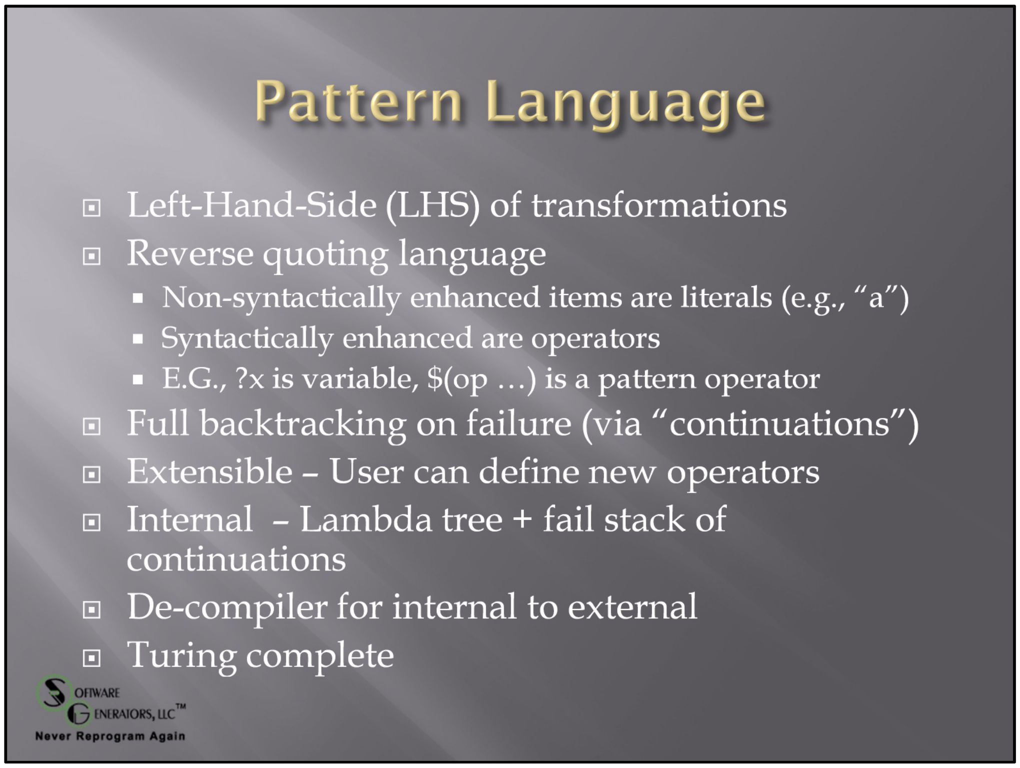


Tools

- Building Logical Architecture
- History Debugger
- Partial Evaluation Engine
- Synchronizing Design Decisions
- Pattern Matching Engine
- Transformation Engine
- Type Inference Engine
- Inference Engine

SOFTWARE GENERATORS, LLC™
Never Reprogram Again

The Pattern Matching engine is one of the primary facilities of the generator and is used widely in transformations, in most Lisp routines in the generator, in the code generator, in the type inference engine and in the logical inference engine. Let's take a look at this facility.



Pattern Language

- Left-Hand-Side (LHS) of transformations
- Reverse quoting language
  - Non-syntactically enhanced items are literals (e.g., “a”)
  - Syntactically enhanced are operators
  - E.G., ?x is variable, $(op …) is a pattern operator
- Full backtracking on failure (via “continuations”)
- Extensible – User can define new operators
- Internal – Lambda tree + fail stack of continuations
- De-compiler for internal to external
- Turing complete

SOFTWARE GENERATORS, LLC™
Never Reprogram Again

(Speaker’s note: This is a build slide. Transitions noted in bold.)

Click.

One of main uses of pattern language is to specify the LHS of transformations. It is also used in almost every major Lisp function within DSLGen™. Some of the LHS of a transformation is written by the domain engineer and some of it is generated automatically by DSLGen™ for its own internal design purposes (e.g., a transformation’s LHS creates bindings for transformation name, home object, phase, the object defining the details of the transformation, etc. as well as introducing some trace machinery). The pattern language is built on top of LISP using Lisp’s character macro feature to “compile” pattern language operators into Lisp structures (e.g., lambda trees).

Click.

The pattern language is a so-called “reverse quoting” language, which means that anything without some special syntax is a literal. Two syntactic enhancements identify pattern language elements – a question mark (?) identifies variables that will be bound to data (e.g., literals or expressions), and a dollar sign ($) identifies pattern operators such as $(pand ?x z), which requires that the next data item be bound to the variable ?x AND that data item must be the literal item “z”. A further operator example is the Por operator, which allows a list of alternative patterns, one of which must match for the Por match to succeed.

Click.

The pattern matcher uses a control structure based on “continuations”, which allows full backtracking. That is, if a pattern match is succeeding and the next sub-pattern match fails on the next data item, the matcher undoes all successful operations back to the last choice point (i.e., the place where there is some alternative sub-pattern that has not been tried but potentially might succeed) and the matching operation is restarted at that choice point with all bindings that were created as a result of successful matches of the prefix data matched up to the choice point. Thus, these matches are “exhaustive” and can only fully fail if there is no possible pathway through the pattern that matches the data the matcher is supposed to match. The data being matched in the DSLGen™ ‘s case is often an Abstract Syntax Tree (AST) or an AST subtree.

Click.

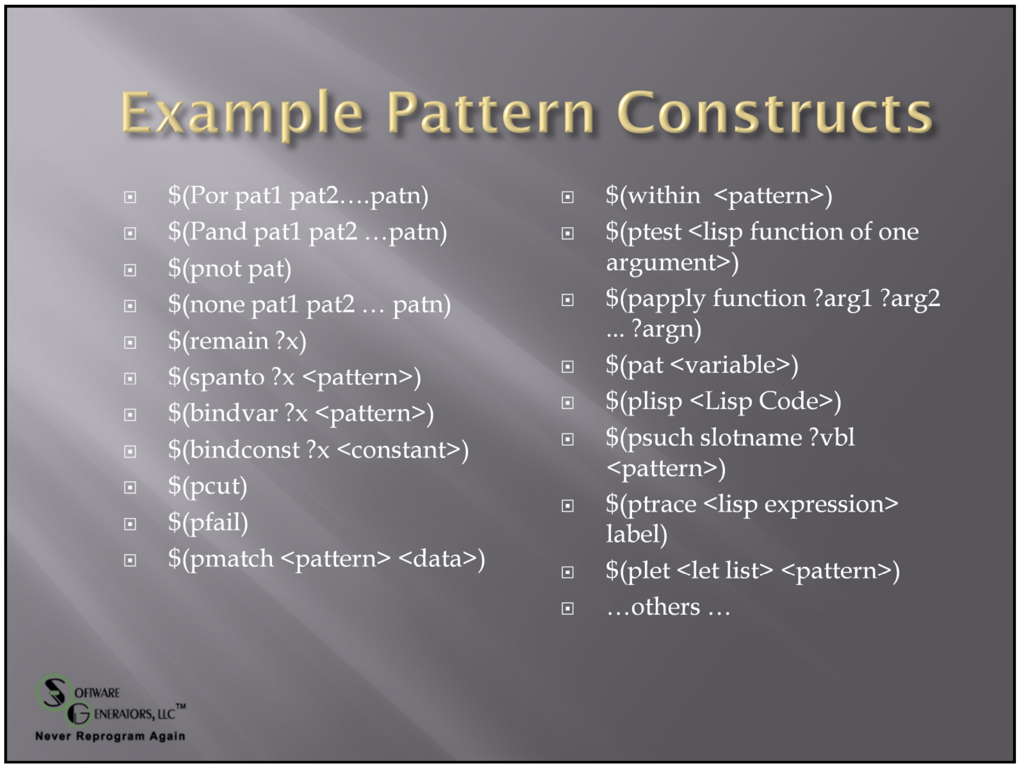
The pattern language has several dozen operators but it is extensible. The Domain Engineer can create new pattern operators using Lisp macros provided in DSLGen™.

Click.

The basic data structures are a “lambda tree” that is the internal form of the pattern and a failure stack that contains continuations for choice points that have not yet been explored. There is a “decompiler” function to express the lambda tree in the form that a pattern writer would use to express the pattern. This decompiler facility is used by a number of tools and inspectors that a Domain Engineer needs to examine internal structures while extending or debugging an addition to DSLGen™.

Click.

The pattern matcher is “Turing complete” meaning that it can compute anything a Turing machine can compute. That is, any computation that can be expressed in a programming language can be expressed in the pattern language.

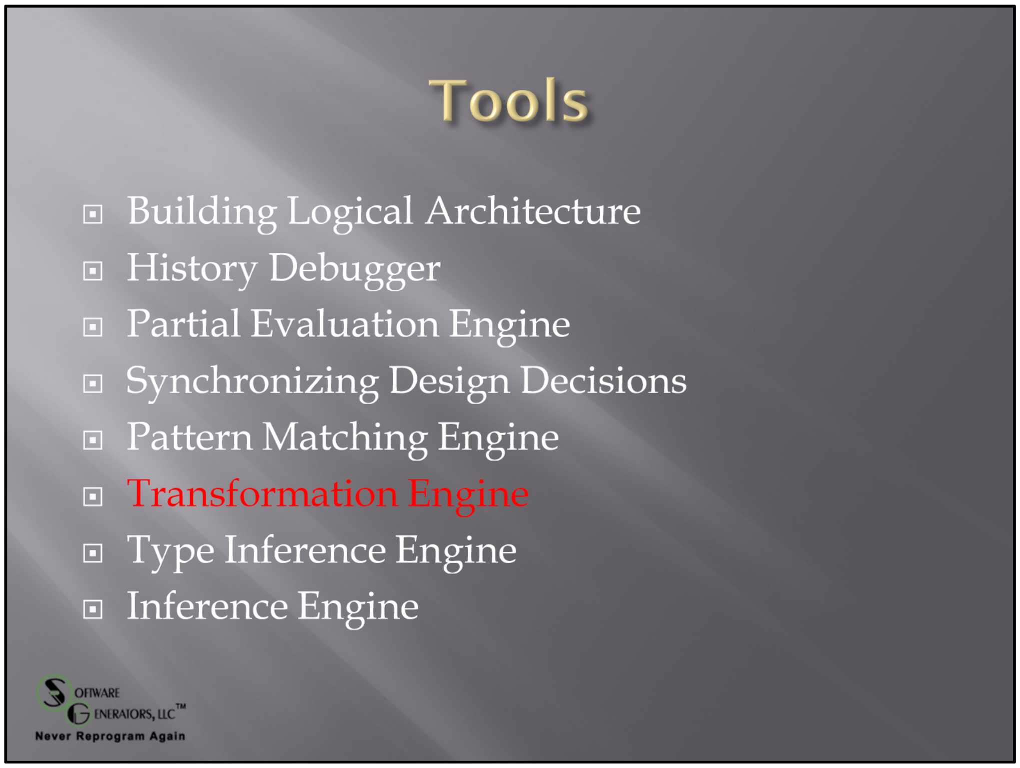


Example Pattern Constructs

- $(Por pat1 pat2….patn)
- $(Pand pat1 pat2 …patn)
- $(pnot pat)
- $(none pat1 pat2 … patn)
- $(remain ?x)
- $(spanto ?x <pattern>)
- $(bindvar ?x <pattern>)
- $(bindconst ?x <constant>)
- $(pcut)
- $(pfail)
- $(pmatch <pattern> <data>)
- $(within <pattern>)
- $(ptest <lisp function of one argument>)
- $(papply function ?arg1 ?arg2 ... ?argn)
- $(pat <variable>)
- $(plisp <Lisp Code>)
- $(psuch slotname ?vbl <pattern>)
- $(ptrace <lisp expression> label)
- $(plet <let list> <pattern>)
- …others …

SOFTWARE GENERATORS, LLC™ Never Reprogram Again

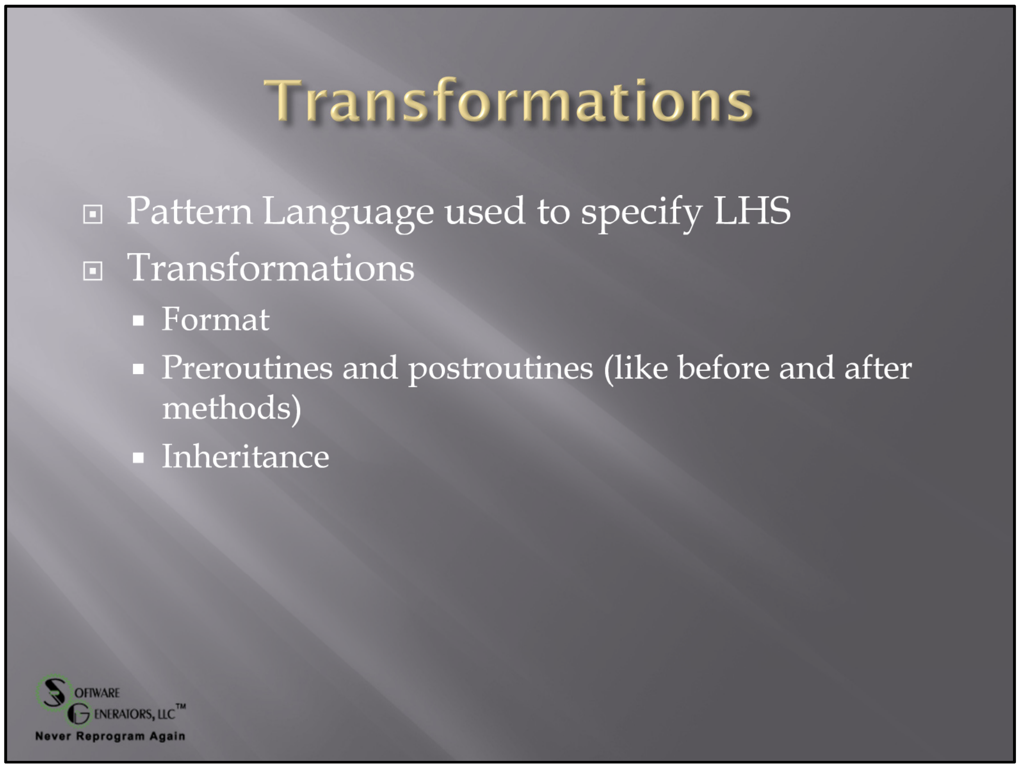
This slide just shows a selection of pattern operators. In addition to the expected operators (e.g., or, and, not and none), there are operators that match an indefinite length list of items (e.g., remain, spanto, spanthru, etc.). There are operators to add user control or management to the match (e.g., bindvar and bindconst to preset certain bindings for the rest of the search), mark and cut to avoid searching paths that are known to be failures, fail and succeed to use knowledge gleaned from the earlier part of the search, various forms of recursive searches (e.g., pmatch, recurse, recurse using this rule, etc.), scoping operators for ?vbls (e.g., plet), various forms for calling Lisp functions (e.g., plisp, papply, ptest), and operators for matching the value of CLOS slots (e.g., psuch).



Tools

- Building Logical Architecture
- History Debugger
- Partial Evaluation Engine
- Synchronizing Design Decisions
- Pattern Matching Engine
- Transformation Engine
- Type Inference Engine
- Inference Engine

SOFTWARE GENERATORS, LLC™
Never Reprogram Again

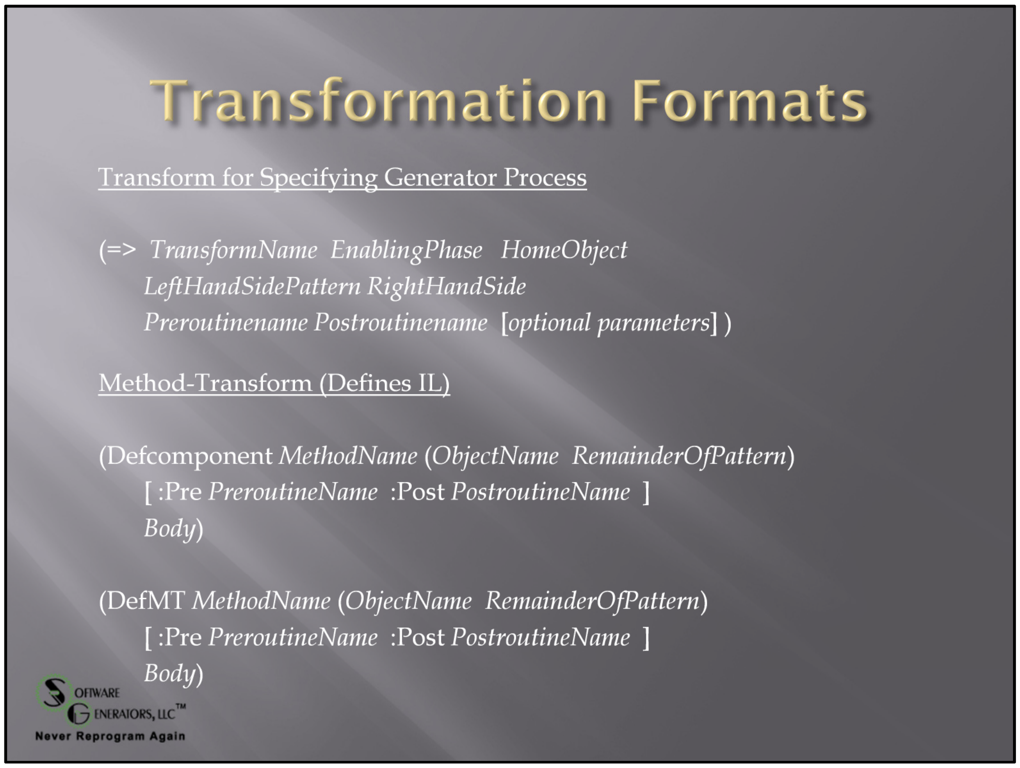


Transformations

- Pattern Language used to specify LHS
- Transformations
  - Format
  - Preroutines and postroutines (like before and after methods)
  - Inheritance

OFTWARE ENERATORS, LLC™
Never Reprogram Again

The following slides provide a little closer look at DSLGen™ transformations.



Transformation Formats

Transform for Specifying Generator Process

```
(=> TransformName  EnablingPhase   HomeObject
      LeftHandSidePattern RightHandSide
      Preroutinename Postroutinename  [optional parameters] )
```

Method-Transform (Defines IL)

```
(Defcomponent MethodName (ObjectName  RemainderOfPattern)
      [ :Pre PreroutineName  :Post PostroutineName  ]
      Body)

(DefMT MethodName (ObjectName  RemainderOfPattern)
      [ :Pre PreroutineName  :Post PostroutineName  ]
      Body)
```

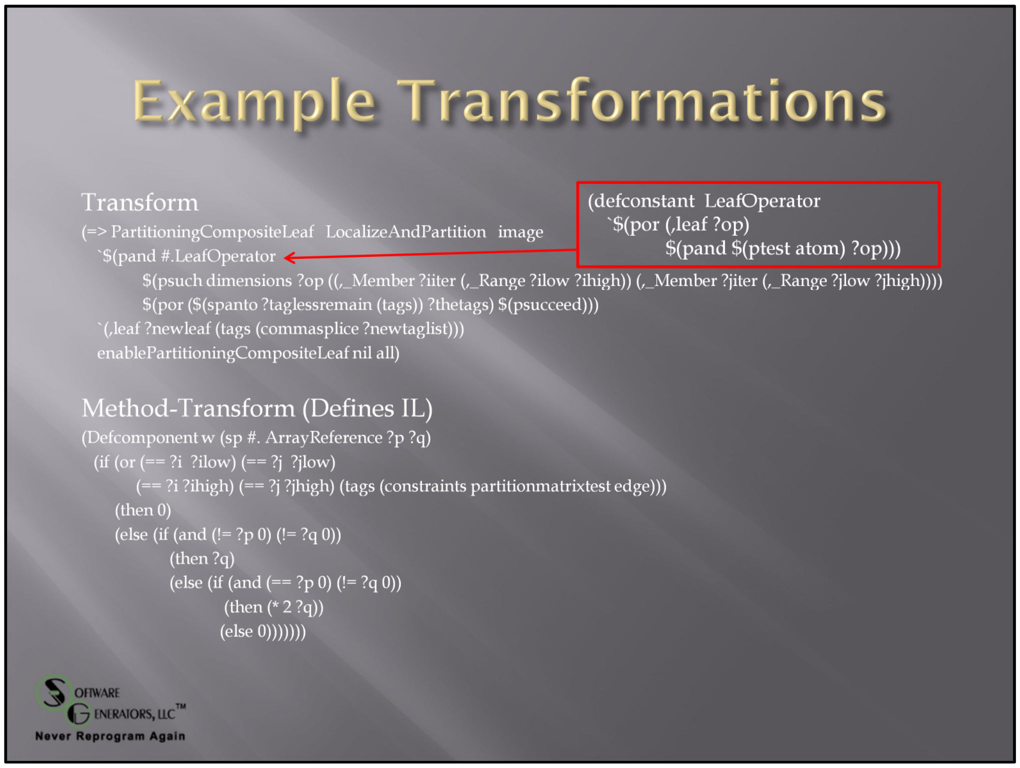
Software Generators, LLC™
Never Reprogram Again

There are two basic kinds of transformations: Process and Method-Transform.

The DSLGen™ generation process is defined by sets of process transformations defined for each declared phase in the generation process. Each such transform is stored as a CLOS object whose name is the triple *TransformName* . *HomeObject* . *EnablingPhase* and it lives on the CLOS type object named *HomeObject.* Most process transforms have preroutines (conventionally named as the transform name prefixed by "enable," although any Lisp symbol is acceptable). The preroutines do bookkeeping chores (e.g., inventing internal names and creating new generator objects) and sometimes extend the bindings list produced by the LHS pattern match.

Method transforms are a specialized version of transforms that are specified like and behave like OO methods. Internally, they are exactly like a process transformation. We have already shown several examples of MTs and described their function at some length. Their LHS pattern is manufactured from *MethodName, ObjectName* and *RemainderOfPattern* specification. Their RHS is the body of the MT definition. They rarely need pre or post routines but they are optionally available. MTs are stored as a CLOS object whose name is the triple *TransformName* . *HomeObject* . Formals and they live on the CLOS type object named *HomeObject.* The phase for which they are enabled is implied to be the phase "Formals" but it is optionally possible to specify some other phase name. However, that is very rarely used.

There are two Method-transform formats accepted. The Defcomponents form is the old format and may possibly be phased out. The newer format is the DefMT format. The minor differences are not relevant at this level of discussion.



Example Transformations

Transform

```
(=> PartitioningCompositeLeaf   LocalizeAndPartition   image
   `$(pand #.LeafOperator
          $(psuch dimensions ?op ((,_Member ?iiter (,_Range ?ilow ?ihigh)) (,_Member ?jiter (,_Range ?jlow ?jhigh))))
          $(por ($(spanto ?taglessremain (tags)) ?thetags) $(psucceed)))
   `(,leaf ?newleaf (tags (commasplice ?newtaglist)))
   enablePartitioningCompositeLeaf nil all)
```

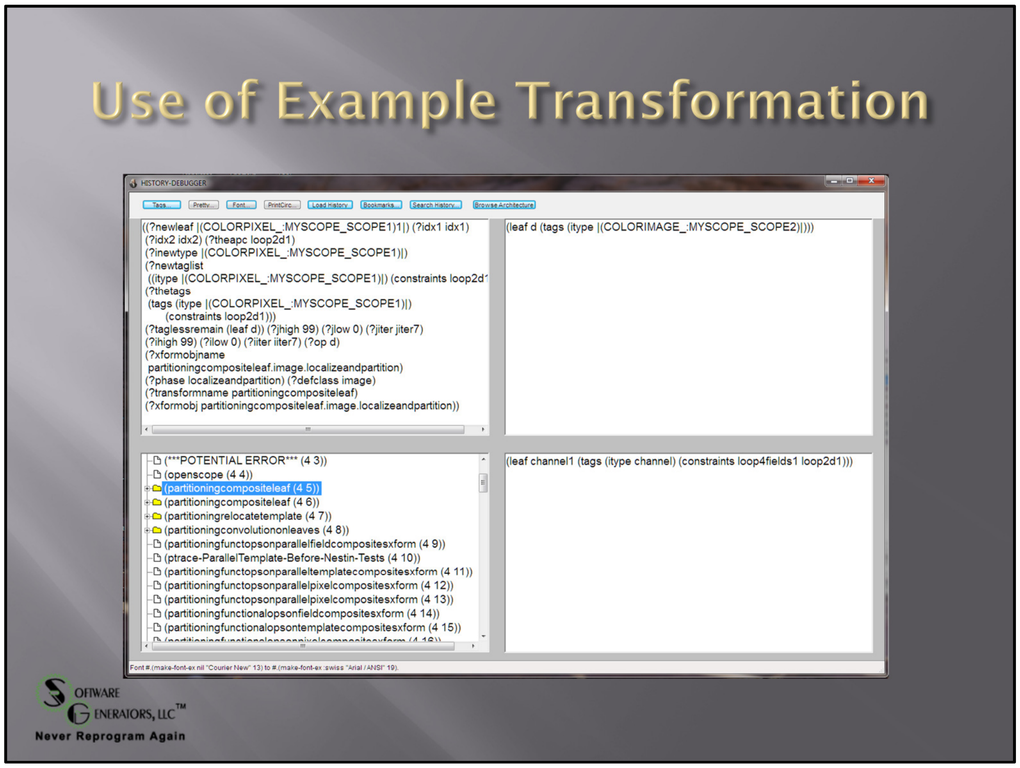
```
(defconstant  LeafOperator
   `$(por (,leaf ?op)
          $(pand $(ptest atom) ?op)))
```

Method-Transform (Defines IL)

```
(Defcomponent w (sp #. ArrayReference ?p ?q)
  (if (or (== ?i  ?ilow) (== ?j  ?jlow)
        (== ?i ?ihigh) (== ?j ?jhigh) (tags (constraints partitionmatrixtest edge)))
     (then 0)
     (else (if (and (!= ?p 0) (!= ?q 0))
              (then ?q)
              (else (if (and (== ?p 0) (!= ?q 0))
                       (then (* 2 ?q))
                       (else 0)))))))
```

We have seen both of these example transforms previously.

Click for build action, which shows the definition of <u>LeafOperator</u>. Explain shared pattern subparts (which uses Lisp's "#." syntax). They are widely used to build complex patterns from simpler pieces.



Use of Example Transformation

```
((?newleaf |(COLORPIXEL_:MYSCOPE_SCOPE1)1|) (?idx1 idx1)
 (?idx2 idx2) (?theapc loop2d1)
 (?inewtype |(COLORPIXEL_:MYSCOPE_SCOPE1)|)
 (?newtaglist
  ((itype |(COLORPIXEL_:MYSCOPE_SCOPE1)|) (constraints loop2d1
 (?thetags
  (tags (itype |(COLORPIXEL_:MYSCOPE_SCOPE1)|)
      (constraints loop2d1)))
 (?taglessremain (leaf d)) (?jhigh 99) (?jlow 0) (?jiter jiter7)
 (?ihigh 99) (?ilow 0) (?iiter iiter7) (?op d)
 (?xformobjname
  partitioningcompositeleaf.image.localizeandpartition)
 (?phase localizeandpartition) (?defclass image)
 (?transformname partitioningcompositeleaf)
 (?xformobj partitioningcompositeleaf.image.localizeandpartition))
```

```
(leaf d (tags (itype |(COLORIMAGE_:MYSCOPE_SCOPE2)|)))
```

```
(***POTENTIAL ERROR*** (4 3))
(openscope (4 4))
(partitioningcompositeleaf (4 5))
(partitioningcompositeleaf (4 6))
(partitioningrelocatetemplate (4 7))
(partitioningconvolutiononleaves (4 8))
(partitioningfunctopsonparallelfieldcompositesxform (4 9))
(ptrace-ParallelTemplate-Before-Nestin-Tests (4 10))
(partitioningfunctopsonparalleltemplatecompositesxform (4 11))
(partitioningfunctopsonparallelpixelcompositesxform (4 12))
(partitioningfunctopsonparallelpixelcompositesxform (4 13))
(partitioningfunctionalopsonfieldcompositesxform (4 14))
(partitioningfunctionalopsontemplatecompositesxform (4 15))
```

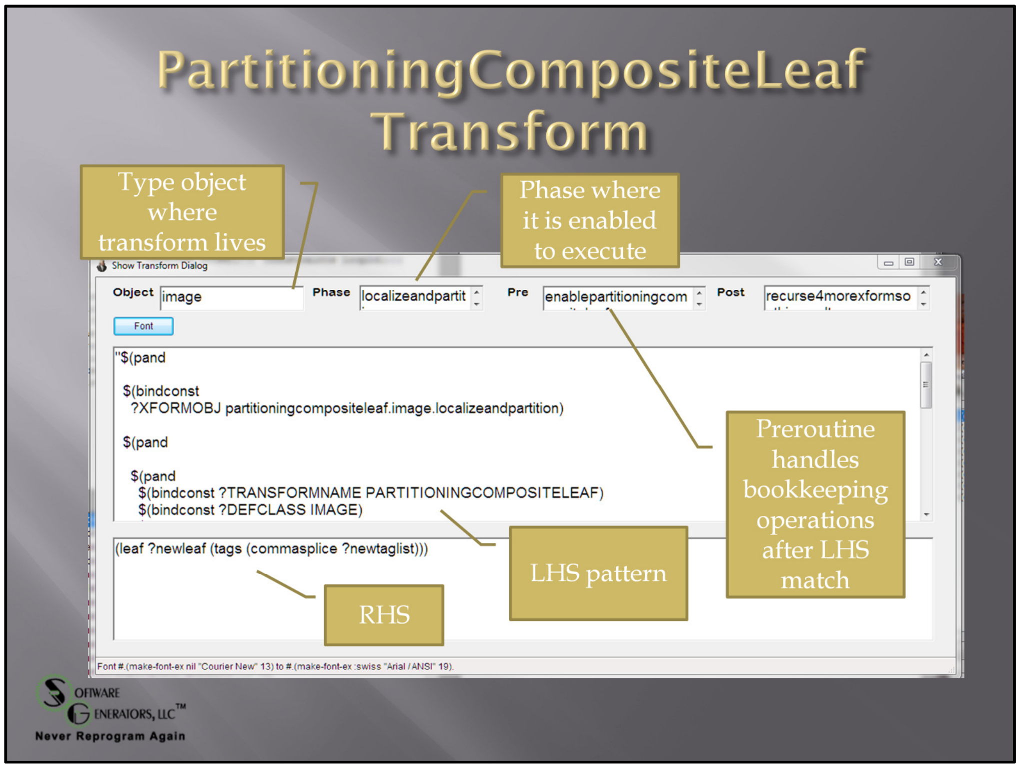
```
(leaf channel1 (tags (itype channel) (constraints loop4fields1 loop2d1)))
```

```
Font #.(make-font-ex nil "Courier New" 13) to #.(make-font-ex :swiss "Arial / ANSI" 19).
```

The selected transform (i.e., partitioningcompositeleaf) is the transform that is doing the first transformation on the image “d”. It introduces indexing variables and loop constraints for the loop over image (i.e., loop2d1) and loop over RGB fields (i.e., loop4fields1). Channel1 is the temporary variable introduced to index the fields.

(Speaker’s note: This is bookmarked in the live example but using the slides is more time efficient.)

PartitioningCompositeLeaf Transform


Type object where transform lives
Phase where it is enabled to execute
Preroutine handles bookkeeping operations after LHS match
LHS pattern
RHS
Show Transform Dialog
Font

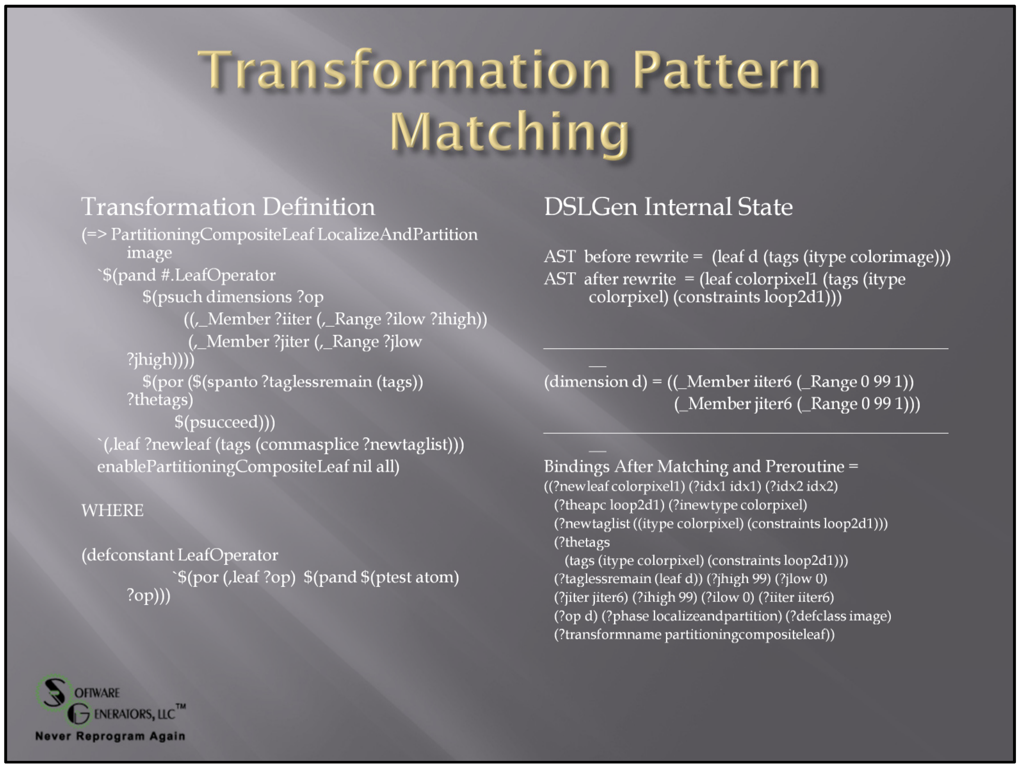
Object image Phase localizeandpartit Pre enablepartitioningcom Post recurse4morexformso

```
"$(pand

 $(bindconst
  ?XFORMOBJ partitioningcompositeleaf.image.localizeandpartition)

 $(pand

  $(pand
   $(bindconst ?TRANSFORMNAME PARTITIONINGCOMPOSITELEAF)
   $(bindconst ?DEFCLASS IMAGE)
```

```
(leaf ?newleaf (tags (commasplice ?newtaglist)))
```

Font #.(make-font-ex nil "Courier New" 13) to #.(make-font-ex :swiss "Arial / ANSI" 19).

This slide is here because this instance shows the specific transformation (i.e., "partitioningcompositeleaf") that was selected in the previous slide. The Domain Engineer (apparently) has double clicked the selected outline item (i.e., (partitioningcompositeleaf (4 6))) in the History Debugger instance shown in the previous slide and thereby produced this dialog instance. This provides the opportunity to further characterize the behavior of preroutines by noting that the values for ?newleaf and ?newtaglist are constructed by the preroutine of this transformation using parts and pieces that were found during the LHS match. They are added to the binding list upon return from the preroutine.



Transformation Pattern Matching

Transformation Definition

```
(=> PartitioningCompositeLeaf LocalizeAndPartition
      image
  `$(pand #.LeafOperator
         $(psuch dimensions ?op
               ((,_Member ?iiter (,_Range ?ilow ?ihigh))
                (,_Member ?jiter (,_Range ?jlow
      ?jhigh))))
         $(por ($(spanto ?taglessremain (tags))
      ?thetags)
              $(psucceed)))
  `(,leaf ?newleaf (tags (commasplice ?newtaglist)))
  enablePartitioningCompositeLeaf nil all)

WHERE

(defconstant LeafOperator
              `$(por (,leaf ?op)  $(pand $(ptest atom)
      ?op)))
```

DSLGen Internal State

```
AST  before rewrite =  (leaf d (tags (itype colorimage)))
AST  after rewrite  = (leaf colorpixel1 (tags (itype
        colorpixel) (constraints loop2d1)))
```

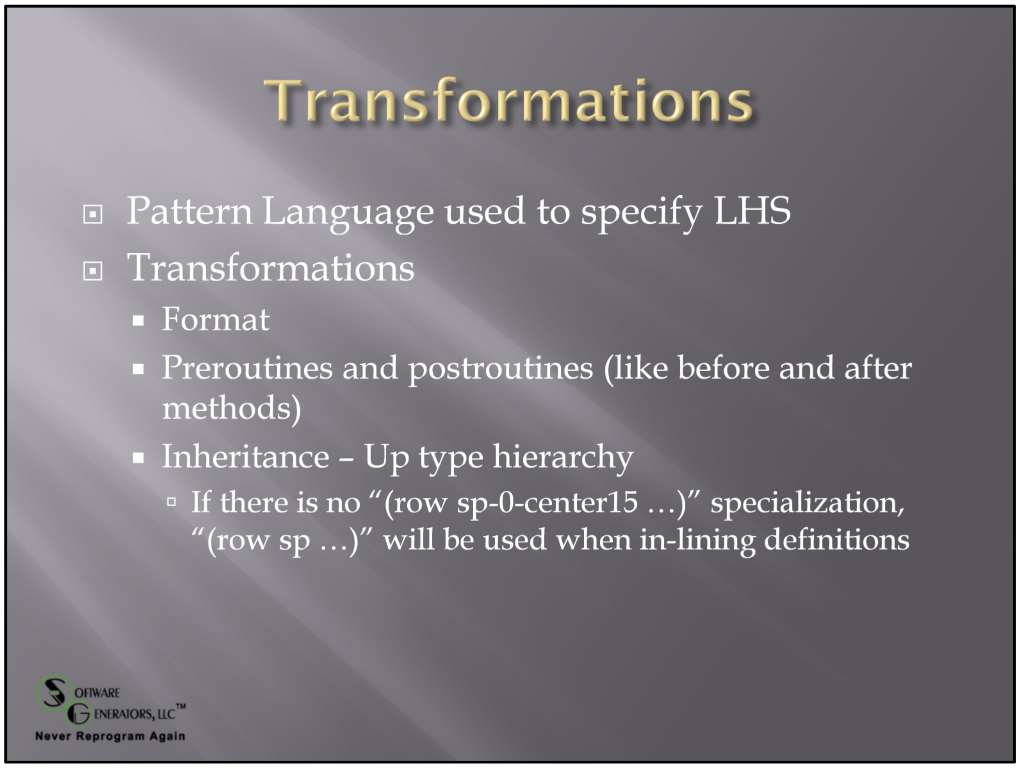
(dimension d) = ((_Member iiter6 (_Range 0 99 1))
(_Member jiter6 (_Range 0 99 1)))

Bindings After Matching and Preroutine =

```
((?newleaf colorpixel1) (?idx1 idx1) (?idx2 idx2)
  (?theapc loop2d1) (?inewtype colorpixel)
  (?newtaglist ((itype colorpixel) (constraints loop2d1)))
  (?thetags
    (tags (itype colorpixel) (constraints loop2d1)))
  (?taglessremain (leaf d)) (?jhigh 99) (?jlow 0)
  (?jiter jiter6) (?ihigh 99) (?ilow 0) (?iiter iiter6)
  (?op d) (?phase localizeandpartition) (?defclass image)
  (?transformname partitioningcompositeleaf))
```

SOFTWARE GENERATORS, LLC™
Never Reprogram Again

Redundant. Skip.



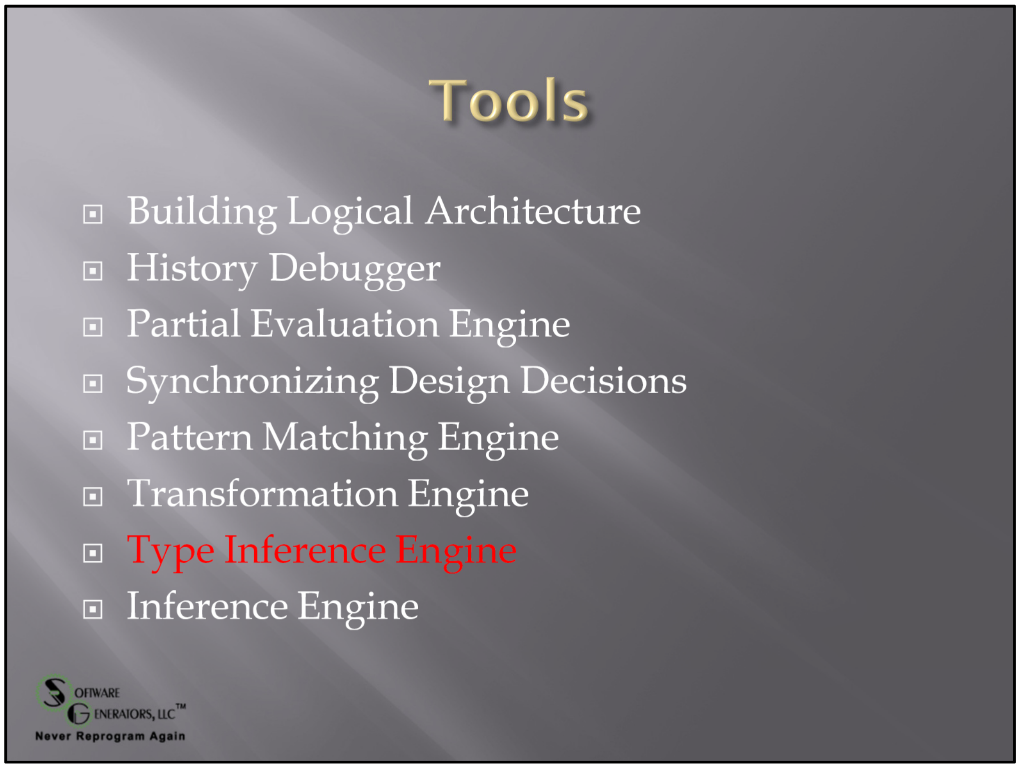
Transformations

- Pattern Language used to specify LHS
- Transformations
  - Format
  - Preroutines and postroutines (like before and after methods)
  - Inheritance – Up type hierarchy
    - If there is no "(row sp-0-center15 …)" specialization, "(row sp …)" will be used when in-lining definitions

SOFTWARE GENERATORS, LLC™
Never Reprogram Again

Talk about the **type hierarchy** and how transformation **inheritance** works. Discuss how this serves specialization of component definitions (i.e., MT-s). Mention that **type inference rules** (although distinct from true transformations) use the same kind of inheritance.

For in depth talks (e.g., a Domain Engineering class), the speaker may want to show the Lisp type tree, if there is enough time (and you are running a live example) just to provide a little context enrichment. Show some transforms relevant to specific types and (perhaps) inspect a raw transform object for concrete grounding. However, for general information talks, such details are best avoided.

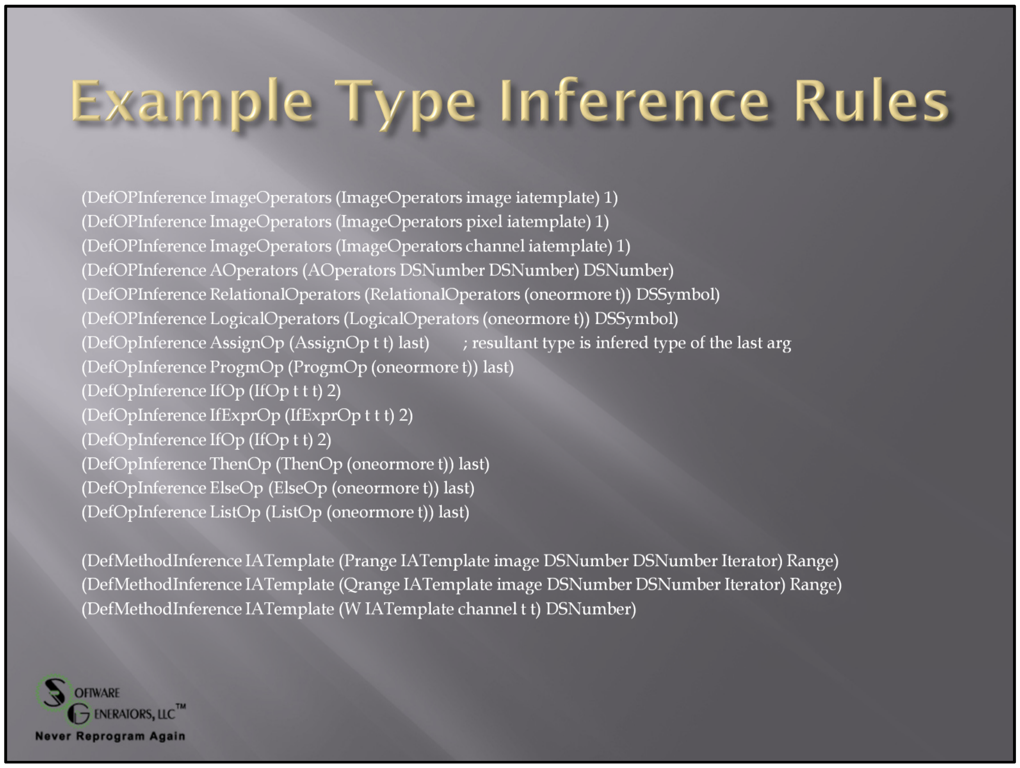


Tools

- Building Logical Architecture
- History Debugger
- Partial Evaluation Engine
- Synchronizing Design Decisions
- Pattern Matching Engine
- Transformation Engine
- Type Inference Engine
- Inference Engine

SOFTWARE GENERATORS, LLC™
Never Reprogram Again

The type inference system is rule based. Let’s look at some examples.

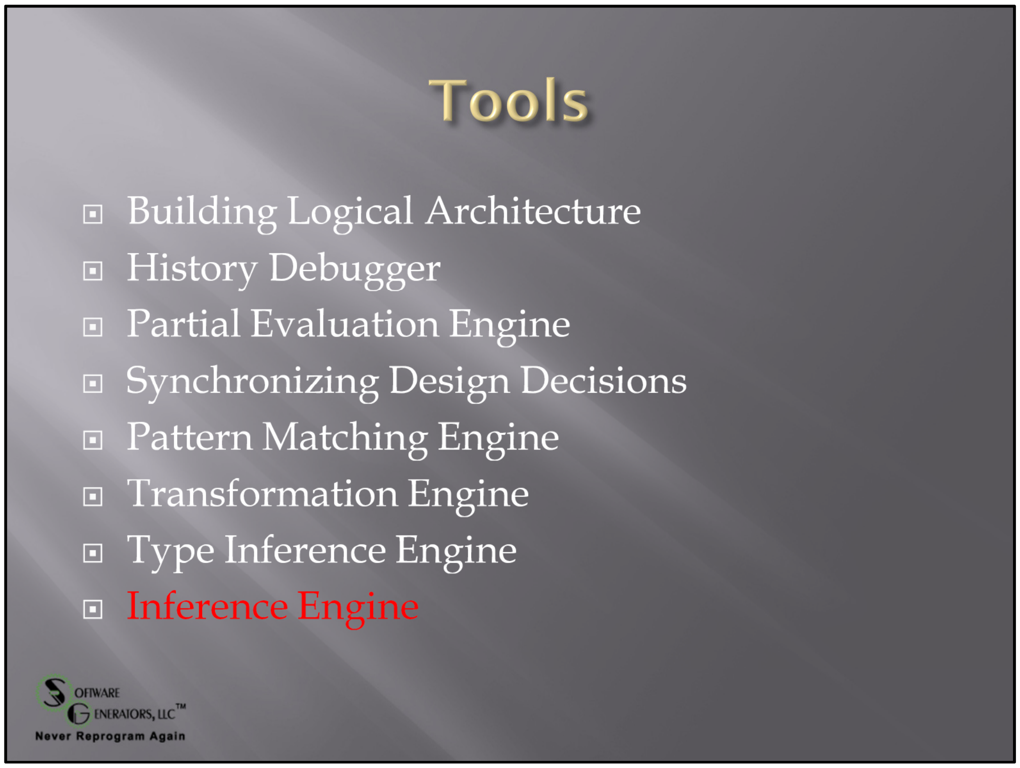


Example Type Inference Rules

```
(DefOPInference ImageOperators (ImageOperators image iatemplate) 1)
(DefOPInference ImageOperators (ImageOperators pixel iatemplate) 1)
(DefOPInference ImageOperators (ImageOperators channel iatemplate) 1)
(DefOPInference AOperators (AOperators DSNumber DSNumber) DSNumber)
(DefOPInference RelationalOperators (RelationalOperators (oneormore t)) DSSymbol)
(DefOPInference LogicalOperators (LogicalOperators (oneormore t)) DSSymbol)
(DefOpInference AssignOp (AssignOp t t) last)        ; resultant type is infered type of the last arg
(DefOpInference ProgmOp (ProgmOp (oneormore t)) last)
(DefOpInference IfOp (IfOp t t t) 2)
(DefOpInference IfExprOp (IfExprOp t t t) 2)
(DefOpInference IfOp (IfOp t t) 2)
(DefOpInference ThenOp (ThenOp (oneormore t)) last)
(DefOpInference ElseOp (ElseOp (oneormore t)) last)
(DefOpInference ListOp (ListOp (oneormore t)) last)

(DefMethodInference IATemplate (Prange IATemplate image DSNumber DSNumber Iterator) Range)
(DefMethodInference IATemplate (Qrange IATemplate image DSNumber DSNumber Iterator) Range)
(DefMethodInference IATemplate (W IATemplate channel t t) DSNumber)
```

Software Generators, LLC™
Never Reprogram Again

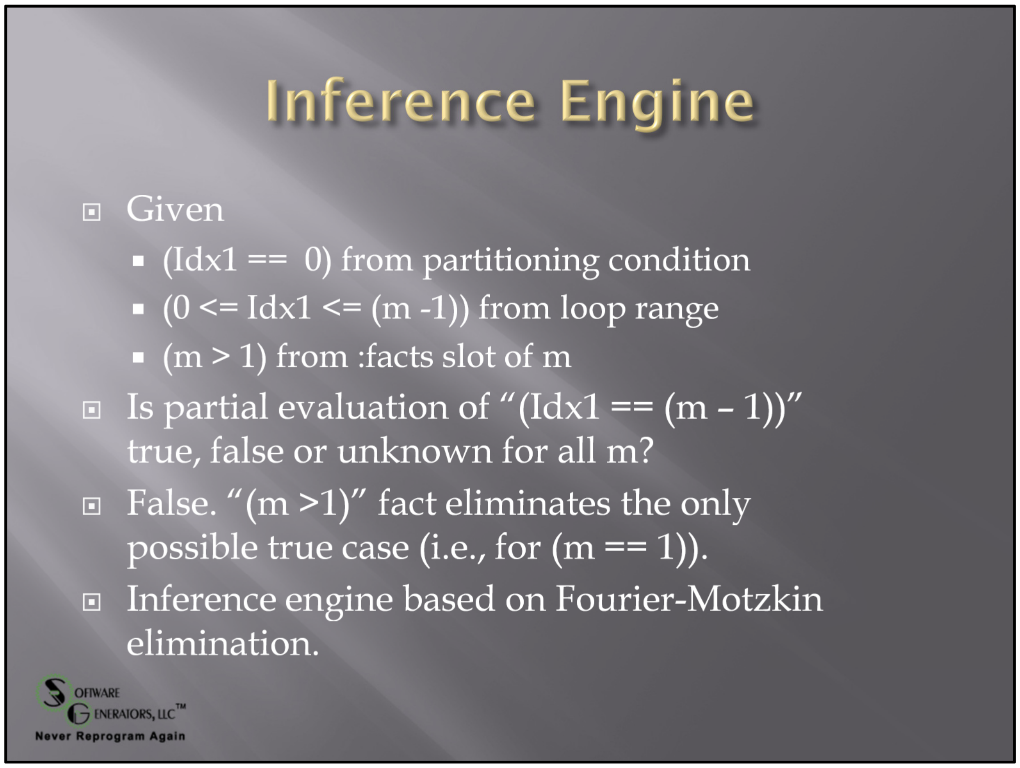
Explain format. Note that defopinference and defMethodInference are macros that compile into patterns. Mention the indefinite length pattern builders oneormore and zeroormore. Illustrate the various forms of specifying the return type -- integer position, explicit type name, “last” for last type matched and others.



Tools

- Building Logical Architecture
- History Debugger
- Partial Evaluation Engine
- Synchronizing Design Decisions
- Pattern Matching Engine
- Transformation Engine
- Type Inference Engine
- Inference Engine

SOFTWARE GENERATORS, LLC™
Never Reprogram Again

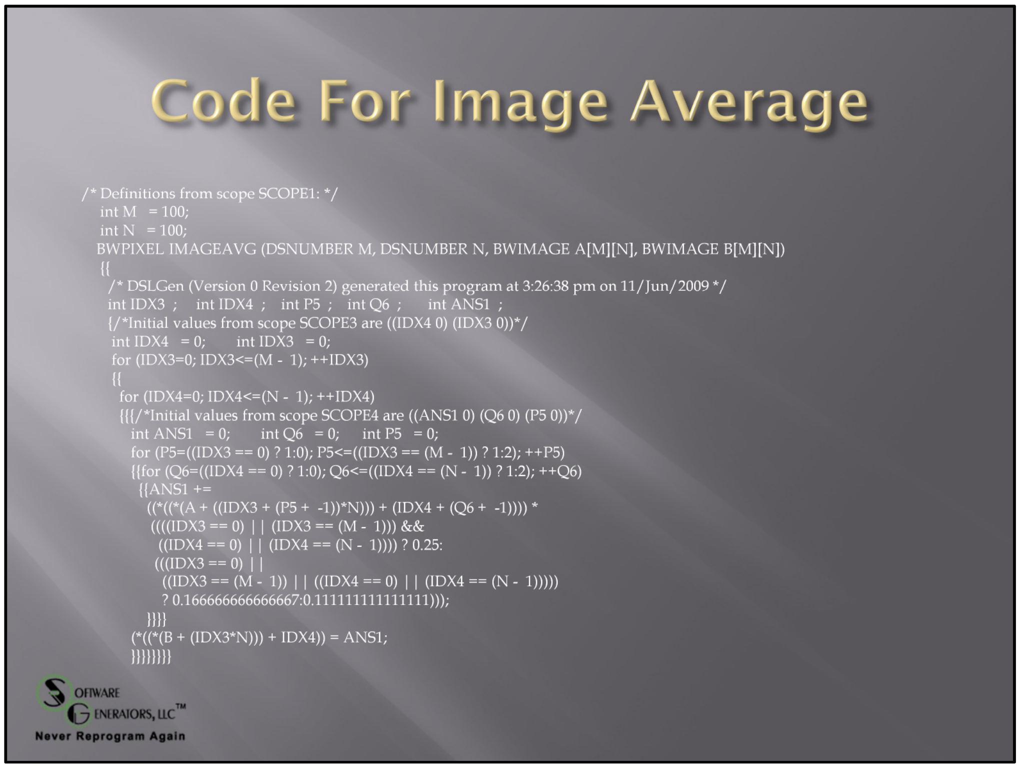


Inference Engine

- Given
  - (Idx1 == 0) from partitioning condition
  - (0 <= Idx1 <= (m -1)) from loop range
  - (m > 1) from :facts slot of m
- Is partial evaluation of "(Idx1 == (m – 1))" true, false or unknown for all m?
- False. "(m >1)" fact eliminates the only possible true case (i.e., for (m == 1)).
- Inference engine based on Fourier-Motzkin elimination.

SOFTWARE GENERATORS, LLC™
Never Reprogram Again

Self explanatory.



Code For Image Average

```
/* Definitions from scope SCOPE1: */
   int M   = 100;
   int N   = 100;
  BWPIXEL IMAGEAVG (DSNUMBER M, DSNUMBER N, BWIMAGE A[M][N], BWIMAGE B[M][N])
   {{
    /* DSLGen (Version 0 Revision 2) generated this program at 3:26:38 pm on 11/Jun/2009 */
    int IDX3  ;    int IDX4  ;    int P5  ;    int Q6  ;       int ANS1  ;
    {/*Initial values from scope SCOPE3 are ((IDX4 0) (IDX3 0))*/
     int IDX4   = 0;        int IDX3   = 0;
     for (IDX3=0; IDX3<=(M -  1); ++IDX3)
     {{
      for (IDX4=0; IDX4<=(N -  1); ++IDX4)
      {{{/*Initial values from scope SCOPE4 are ((ANS1 0) (Q6 0) (P5 0))*/
        int ANS1   = 0;       int Q6   = 0;     int P5   = 0;
        for (P5=((IDX3 == 0) ? 1:0); P5<=((IDX3 == (M -  1)) ? 1:2); ++P5)
        {{for (Q6=((IDX4 == 0) ? 1:0); Q6<=((IDX4 == (N -  1)) ? 1:2); ++Q6)
         {{ANS1 +=
           ((*((*(A + ((IDX3 + (P5 +  -1))*N))) + (IDX4 + (Q6 +  -1)))) *
            ((((IDX3 == 0) | | (IDX3 == (M -  1))) &&
              ((IDX4 == 0) | | (IDX4 == (N -  1)))) ? 0.25:
             (((IDX3 == 0) | |
               ((IDX3 == (M -  1)) | | ((IDX4 == 0) | | (IDX4 == (N -  1)))))
               ? 0.166666666666667:0.111111111111111)));
           }}}}
        (*((*(B + (IDX3*N))) + IDX4)) = ANS1;
        }}}}}}}}
```

OFTWARE
ENERATORS, LLC™
Never Reprogram Again

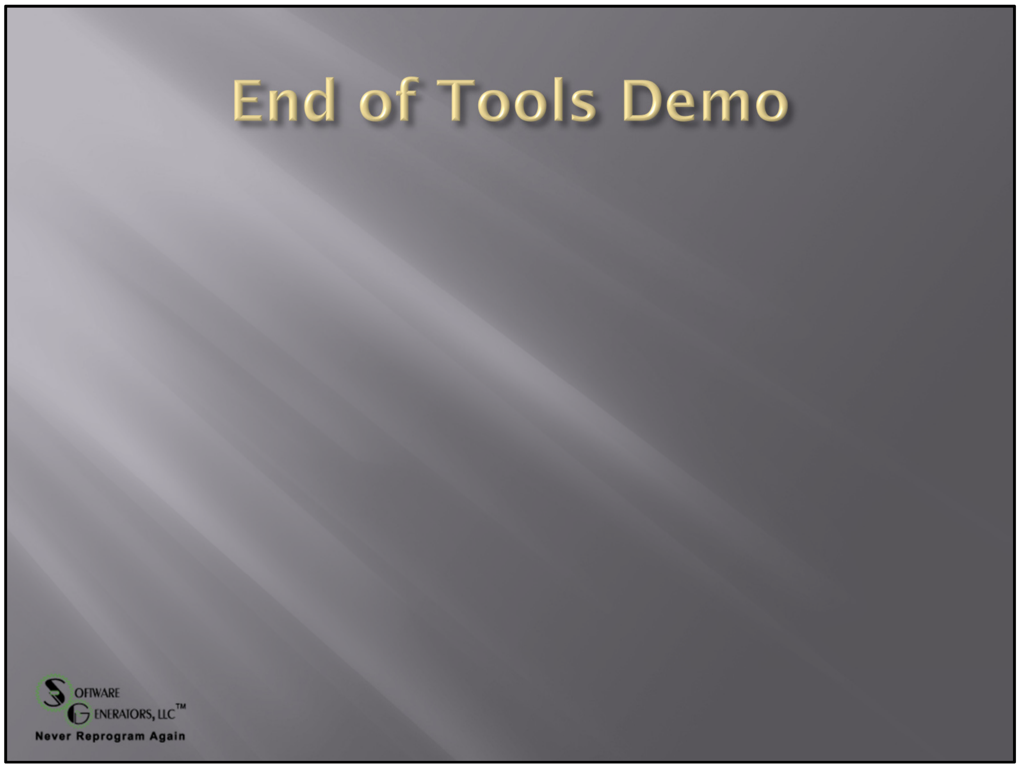
Old example for a different computation.

SOFTWARE GENERATORS, LLC™

Never Reprogram Again

SOFTWARE GENERATORS, LLC™

Never Reprogram Again



End of Tools Demo

SOFTWARE GENERATORS, LLC™
Never Reprogram Again