CISC 858: Team “iCompile”:

Phase 4 Documentation

Mike Michaud

Tammy Jiang

1. Objectives of Phase 4 of the CISC 458 Project: Code Generation

In phase 4 of the CISC 458 project, the Code Generation phase of the PT Pascal compiler was modified to support JT.

The following modifications were made to the Code Generation phase of the PT Pascal compiler in order to conform to previous changes to the Scanner, Parser and Semantic Phase for JT:

1. Synchronize tokens between Semantic Phase output and Code Generator input;
2. Add trap codes for the new JT string handling features to link to the PT run time string operation traps;
3. Remove all Char semantic operations and add new String semantic operations;
4. Add handling for classes, which imply that declarations and statements may be mixed within the input token stream;
5. Add handling of the new JT String operations;
6. Add subscripting of String arrays (including size 1024;
7. Add support for empty Strings (zero length);
8. Add Code Templates for String operations;
9. Add handling of Loop statements (replacing While and Repeat/Until statements);
10. Add handling of function returns (values and expressions); and
11. Add handling of function calls to user-defined functions.

2. Current State of the Source Code

“Phase 4” of the source code changes (detailed above) are now complete. This phase includes changes to the Code Generator (coder.ssl and coder.pt).

Test cases that prove the source code changes (detailed above), are included with the project submission for this phase. Each test case is described in a separate file that includes the source file, details the purpose, expected/actual result and the conclusion (pass/fail) of the test case.

3. Summary of Source Code Changes

The following section details the source code changes and in which context the changes were made. For each feature change, the changes in each file are detailed in a “pseudo-diff” style.

***Differences are noted between the original PT files and the JT Phase 4 submission.

Raw “diff” files can be found in the project submission directory as:

- “`coder.pt_diff_out`”
- “`coder.ssl_diff_out`”

3.1 Tokens Added / Removed / Changed

The changes here were made to make the code generator conform to the semantic output tokens from phase 3.

- FILE: "`coder.ssl`" (new = “<”, old = “>”)

```
32,136c32,88
<       tMultiply
<       firstInputToken = tMultiply
<       tDivide
<       tModulus
<       tAdd
<       tSubtract
<       tEQ
<       tNE
<       tGT
<       tGE
```

```
<       tLT
<       tLE
<       tAnd
<       tInfixAnd
<       tOr
<       tInfixOr
<       tNegate
<       tNot
<       tChr
<       tOrd
<       tEoln
<       tEOF
<       tVarParm
<       tFetchAddress
<       tFetchInteger
<       tFetchString                %%%%%%%%%%% CHANGED
<       tFetchBoolean
<       tAssignBegin
<       tAssignAddress
<       tAssignInteger
<       tAssignString               %%%%%%%%%%% CHANGED
<       tAssignBoolean
<       tStoreParmAddress
<       tStoreParmInteger
<       tStoreParmString            %%%%%%%%%%% CHANGED
<       tStoreParmBoolean
<       tSubscriptBegin
<       tSubscriptAddress
<       tSubscriptInteger
<       tSubscriptString            %%%%%%%%%%% CHANGED
<       tSubscriptBoolean
<       tArrayDescriptor
<       tFileDescriptor
<       tIfBegin
<       tIfEnd
<       tCaseBegin
< %%%%%%%%%%% CHANGED

< %     tWhileBegin
< %     tRepeatBegin
< %     tRepeatControl
< %%%%%%%%%%%
<       tLoopBegin
<       tLoopBreakWhen
< %%%%%%%%%%%
<       tCallBegin
<       tParmEnd
<       tProcedureEnd
<       tWriteBegin
<       tReadBegin
<       tTrapBegin
<       tWriteEnd
<       tReadEnd
< %%%%%%%%%%% ADDED
<       tCaseElse
<       tConcatenate
<       tSubstring
<       tLength
<       tStringEqual
< %%%%%%%%%%%
< %%%%%%%%%%% ADDED 858
<       tFunctionResult
```

```
< %%%%%%%%%%
<
<       % Compound T-codes are those that take operands
<       tLiteralAddress
<       firstCompoundToken = tLiteralAddress
<       tLiteralInteger
< %%%%%%%%%%% DELETED
< %     tLiteralChar
< %%%%%%%%%%
<       tLiteralBoolean
<       tLiteralString
< %%%% MPM 2th April: In order to keep consistency with semantic phase.
< %%%% we chose not to delete tStringDescriptor and tSkipString from
< %%%% the input but delete them producedures.
<       tStringDescriptor
<       tSkipString

< %%%% keep them to adjust semantic phase.%%%%%
<       tIfThen
<       tIfMerge
<       tCaseSelect
<       tCaseMerge
<       tCaseEnd
< %%%%%%%%%%% CHANGED
< %     tWhileTest
< %     tWhileEnd
< %     tRepeatTest
< %%%%%%%%%%
<       tLoopTest
<       tLoopEnd
<       tSkipProc
<       tCallEnd
<       tLineNumber
<       tTrap
<       lastCompoundToken = tTrap

>       tMultiply
>       firstInputToken = tMultiply
>       tDivide
>       tModulus
>       tAdd
>       tSubtract
>       tEQ
>       tNE
>       tGT
>       tGE
>       tLT
>       tLE
>       tAnd
>       tInfixAnd
>       tOr
>       tInfixOr
>       tNegate
>       tNot
>       tChr
>       tOrd
>       tEoln
>       tEOF
>       tVarParm
>       tFetchAddress
>       tFetchInteger
>       tFetchChar
```

```
>       tFetchBoolean
>       tAssignBegin
>       tAssignAddress
>       tAssignInteger
>       tAssignChar
>       tAssignBoolean
>       tStoreParmAddress
>       tStoreParmInteger
>       tStoreParmChar
>       tStoreParmBoolean
>       tSubscriptBegin
>       tSubscriptAddress
>       tSubscriptInteger
>       tSubscriptChar
>       tSubscriptBoolean
>       tArrayDescriptor
>       tFileDescriptor
>       tIfBegin
>       tIfEnd
>       tCaseBegin
>       tWhileBegin
>       tRepeatBegin
>       tRepeatControl
>       tCallBegin
>       tParmEnd
>       tProcedureEnd
>       tWriteBegin
>       tReadBegin
>       tTrapBegin
>       tWriteEnd
>       tReadEnd
138,139c90,114
<       tEndOfFile
<       lastInputToken = tEndOfFile;


---
>       % Compound T-codes are those that take operands
>       tLiteralAddress
>       firstCompoundToken = tLiteralAddress
>       tLiteralInteger
>       tLiteralChar
>       tLiteralBoolean
>       tLiteralString
>       tStringDescriptor
>       tSkipString
>       tIfThen
>       tIfMerge
>       tCaseSelect
>       tCaseMerge
>       tCaseEnd
>       tWhileTest
>       tWhileEnd
>       tRepeatTest
>       tSkipProc
>       tCallEnd
>       tLineNumber
>       tTrap
>       lastCompoundToken = tTrap
>
>       tEndOfFile
>       lastInputToken = tEndOfFile;
```

3.2 Add Trap Codes and String Sizes

These changes add new trap codes required by the String routines that are called by the JT compiler in the PT run-time library operations.

- FILE: "coder.ssl" (new = “<”, old = “>”)

```
156,157d130
< %%%%%%%% MJM 25Mar16: Add value 10 = ten
<       ten = 10
174a148
>       trWriteString = 7
175a150
>       trWriteChar = 9
176a152
>       trReadChar = 11

182,189d157
<       pttrap101 = 101 %AssignString
<       pttrap102 = 102 %ChrString
<       pttrap103 = 103 %Concatenate
<       pttrap104 = 104 %Substring
<       pttrap105 = 105 %Length
<       pttrap106 = 106 %StringEqual
<       pttrap107 = 107 %ReadString
<       pttrap108 = 108 %WriteString
193,195c161
<       word = 2  % 4 bytes on x86
< %%%%%%%% MJM 25Mar16: Add string kind
<       string = 3;
---
>       word = 2; % 4 bytes on x86
```

3.3 Change Char Semantic Operations to String Semantic Operations

Changes listed here removed all Char and semantic operations and added new String semantic operations. In addition, the oOperandPushStringDescriptor operations was renamed to oOperandPushString for consistency with the rest of the code.

- FILE: "coder.ssl" (new = “<”, old = “>”)

```
< %%%%%%%% MJM 25Mar16: Delete oOperandPushChar : no longer used
< %     oOperandPushChar
< %     % Push an entry with the value of the last accepted character
< %     % (tLiteralChar tcode), length byte and manifest addressing mode.
---
>       oOperandPushChar
```

```
>      % Push an entry with the value of the last accepted character
>      % (tLiteralChar tcode), length byte and manifest addressing mode.

312,313c277
< %%%%%%%% MJM 25Mar16: Change name of oOperandPushStringDescriptor
<      oOperandPushString
---
>      oOperandPushStringDescriptor

1620c1473,1474
<             oOperandPushString % MJM: Changed frm oOperandPushStringDescriptor
---
>             tStringDescriptor
>             oOperandPushStringDescriptor
```

- FILE: "`coder.pt`" (new = “<”, old = “>”)

```
< {%%%%%%%% MJM 25Mar16: Change name of OperandPushStringDescriptor }
<      procedure OperandPushString;
---
>      procedure OperandPushStringDescriptor;
```

3.4 Handle Statements in Class

These changes added handling for classes, which imply that declarations and statements may be mixed within the input token stream.

- FILE: "`coder.ssl`" (new = “<”, old = “>”)

```
789,811d746
<
< %%%%%%%% MJM 25Mar16: Because we now have classes, statement t-codes
< %%%%%%%%                 can now be included within declarations in the
< %%%%%%%%                 t-code stream from the semantic analyser.
< %%%%%%%%                 Therefore, all t-codes from the @Statements rule
< %%%%%%%%                 need to be handled here.
<      | tAssignBegin:
<            @AssignStmt
<      | tCallBegin:
<            @CallStmt
<      | tIfBegin:
<            @IfStmt
<      | tLoopBegin:
<            @LoopStmt
<      | tCaseBegin:
<            @CaseStmt
<      | tWriteBegin:
<            @WriteProc
<      | tReadBegin:
<            @ReadProc
<      | tTrapBegin:
<            @TrapStmt
<
```

3.5 Handle New JT String operations

Change handling of Char into String. This includes deleting handling of t-codes tLiteralChar, and adding handling tLiteralString. It also includes changes to “coder.pt” to remove call to AssertTempsAreAllFree and remove support for tSkipString and tStringDescriptor, tWhileTest.

- FILE: "`coder.ssl`" (new = “<”, old = “>”)

```
< %%%%%%%% MJM 21Mar16: change Char to String, and separate into 2 cases:
< %%%%%%%%                 1. a new one for tStoreParmString
< %%%%%%%%                 2. reuse existing one for tStoreParmBoolean
<             | tStoreParmString:
<                   oOperandSwap               % ... formal, 8(%ebp)
<                   oOperandPushCopy           % ... formal, 8(%ebp), 8(%ebp)
<                   oOperandSwapLeftAndDest    % ... 8(%ebp), formal, 8(%ebp)
<                   oOperandSetLength(string)
<

< %%%%%%%% MJM 25Mar16: Delete: we no longer handle tLiteralChar
< %    | tLiteralChar:
< %          oOperandPushChar
<
---
>      | tLiteralChar:
>            oOperandPushChar

1608,1612c1461,1465
<
< %%%%%%%% MJM 25Mar16: Handle tLiteralString in the context of an
< %%%%%%%%                 operand of an expression.
< %%%%%%%%                 Merged old tStringDescriptor and tSkipString
<      | tLiteralString:
---
>      | tStringDescriptor:
>            oOperandPushStringDescriptor
>            % Get a string literal's address
>            @EmitStringDescriptor            %      lea    sNNN, %T
>      | tSkipString:
1615a1469
>            tLiteralString

1753,1763c1553,1554
<
< %%%%%%%% MJM 25Mar16: tSubscriptChar replaced with tSubscriptString
< %          | tSubscriptChar:
< %                @OperandSubscriptCharPop
<            | tSubscriptString:
<                  %%%%%%%% MJM 26Mar16: Note that this behaves just like
<                  %%%%%%%%            OperandSubscriptIntegerPop
<                  %%%%%%%%            (i.e. tSubscriptInteger), but
<                  %%%%%%%%            with scaling factor of 1024 vs 4
<                  @OperandSubscriptStringPop
<
---
>            | tSubscriptChar:
>                  @OperandSubscriptCharPop
```

```
1772,1774c1563
<
< %%%%%%%% MJM 25Mar16: Remove tFetchString (was tFetchChar) from case
<       | tFetchBoolean:
---
>       | tFetchChar, tFetchBoolean:


2356,2372c1944,1945
<
< %%%%%%%% MJM 25Mar16: Delete: we no longer handle tLiteralChar
< %     | tLiteralChar:
< %           oOperandPushChar
<
< %%%%%%% MJM 25Mar16: Handle tLiteralString instead
<       | tLiteralString:
<             % Emit string literal to data area
<             oEmitNone(iData)                  %       .data
<             % Emit the string
<             oEmitString                       % sNNN: .asciz  "SSSSS"
<             oEmitNone(iText)                  %       .text
<
<             % Get the string literal's address
<             oOperandPushString
<             @EmitStringDescriptor             %       lea    sNNN, %T
<
---
>       | tLiteralChar:
>             oOperandPushChar

2378,2409c1951,1953
<
< %%%%%%%% MJM 25Mar16: Deleted, as tAssignChar is no longer used,
< %     | tAssignChar:
< %           @OperandAssignCharPopPop
< %           >
<

2518,2529c2034,2040
< %%%%%%%% MJM 25Mar16: tAssignChar no longer used, and in this context
< %%%%%%%%                 tChr is not followed by a tAssign* tcode any more
< %%%%%%%%                 and tChr only takes the first character of the string.
< %           [
< %           | tAssignChar:
< %                 @OperandChrAssignPopPop
< %                 >
< %           | *:
< %                 @OperandChr
< %           ]
<             @OperandChr
<
---
>             [
>             | tAssignChar:
>                   @OperandChrAssignPopPop
>                   >
>             | *:
>                   @OperandChr
>             ]


2531,2541c2042,2048
```

```
< %%%%%%%% MJM 25Mar16: tAssignInteger is no longer used in this context
< %%%%%%%%                 as tOrd only takes the first character of the string.
< %           [
< %           | tAssignInteger:
< %                 @OperandOrdAssignPopPop
< %                 >
< %           | *:
< %                 @OperandOrd
< %           ]
<             @OperandOrd
<
---
>             [
>             | tAssignInteger:
>                   @OperandOrdAssignPopPop
>                   >
>             | *:
>                   @OperandOrd
>             ]

3351,3368c2573
< %%%%%%%% MJM 25Mar16: OperandChrAssignPopPop is no longer used; this
< %%%%%%%%                 optimization doesn't make sense for strings.
< %OperandChrAssignPopPop:
< %    % Assume operand's value is in range
< %    % x := y;  x byte, y word
< %    @OperandForceIntoTemp      %      movl   y, %T
< %    oOperandSetLength(byte)  % (treat word in reg as byte to get
< %                             %  the low order byte of the word)
< %    oEmitDouble(iMov)                %      movb   %T, x
< %    @OperandPopAndFreeTemp
< %    @OperandPopAndFreeTemp;
<
```

- FILE: "`coder.pt`" (new = “<”, old = “>”)

```
1590,1592c1559
< {%%%%%%%% MJM 21Mar16: Changed tStringDescriptor to tLiteralString
< %%%%%%%%}
<             Assert ( (compoundToken = tLiteralString), assert28);
---
>             Assert ( (compoundToken = tStringDescriptor), assert28);

1594c1561
<       end { OperandPushString };
---
>       end { OperandPushStringDescriptor };

1894,1900c1854,1856
< {%%%%%%%% MJM 21Mar16: removed tLiteralChar, tStringDescriptor
< %%%%%%%%%                       tSkipString, tWhileTest,
< %%%%%%%%%                       tWhileEnd, tRepeatTest
< %%%%%%%%%                 and added tLoopTest, tLoopEnd.
< %%%%%%%%}
<                           tLiteralAddress, tLiteralInteger, tLiteralBoolean,
<                           tCallEnd,
---
>                           tLiteralAddress, tLiteralInteger, tLiteralChar,
>                           tLiteralBoolean, tStringDescriptor, tSkipString,
```

```
>                           tWhileTest, tWhileEnd, tRepeatTest, tCallEnd,
1902c1858
<                           tLoopTest, tLoopEnd, tCaseSelect, tCaseMerge:
---
>                           tCaseSelect, tCaseMerge:

2309d2251
< {%%%%%%%% MJM 25Mar16: Delete oOperandPushChar : no longer used
2316d2257
< %%%%%%%%}
2368,2370c2309,2310
< {%%%%%%%% MJM 25Mar16: Change name of oOperandPushStringDescriptor }
<                   oOperandPushString:
<                           OperandPushString;
---
>                   oOperandPushStringDescriptor:
>                           OperandPushStringDescriptor;

2257,2260c2203
< {%%%%%%%% MJM 21Mar16: remove call to AssertTempsAreAllFree, as we now
< %%%%%%%%%                use string traps
< %%%%%%%%}
< {%%%%%%%%                       AssertTempsAreAllFree %%%%%%%%}
---
>                           AssertTempsAreAllFree
```

3.6 Handle String Subscripting and Size

The changes here handle string subscripting. The string size is also changed to 1024

- FILE: "`coder.ssl`" (new = “<”, old = “>”)

```
1873,1887d1626
< %%%%%%%% MJM 25Mar16: Used by OperandSubscriptStringPop
< %%%%%%%%                 for tSubscriptString***
< %%%%%%%%                 Note that this behaves just like
< %%%%%%%%                 OperandSubscriptNonManifestIntegerPop
< %%%%%%%%                 (i.e. tSubscriptInteger), but
< %%%%%%%%                 with scaling factor of 1024 vs 4
< OperandSubscriptNonManifestStringPop:
<       [ oOptionTestChecking
<       | yes:
<             @OperandCheckedSubscriptNonManifestStringPop
<       | *:
<             @OperandUncheckedSubscriptNonManifestStringPop
<       ];
<
<

1970,2057d1708
< %%%%%%%% MJM 25Mar16: Used by OperandSubscriptStringPop
< %%%%%%%%                 for tSubscriptString***
< %%%%%%%%                 Note that this behaves just like
< %%%%%%%%                 OperandSubscriptNonManifestIntegerPop
< %%%%%%%%                 (i.e. tSubscriptInteger), but
< %%%%%%%%                 with scaling factor of 1024 vs 4
```

```
< OperandCheckedSubscriptNonManifestStringPop:
<       % Default bounds checking subscript operation
<
<       % Get subscript                                 % ... arraydesc, subscript
<       @OperandForceIntoTemp                           % ... arraydesc, %T

<
<       % Check range if checking, otherwise don't bother
<       [ oOptionTestChecking
<       | yes:
<               oOperandSwap                    % ... %T, arraydesc
<               @OperandPushArrayUpperBound     % ... %T, arraydesc, upper
<               oOperandSwapLeftAndDest         % ... arraydesc, %T, upper
<               @EmitCmp                        %       cmpl   upper, %T
<               oOperandPushJumpCondition(iJle) % if subscript <= upper
<               oEmitConditionalForwardBranch   %       jle    fNN
<               oFixPushLastAddress
<               oOperandPop                     % ... arraydesc, %T, upper
<               oOperandPop                     % ... arraydesc, %T
<
<               oOperandSwap                    % ... %T, arraydesc
<               @OperandPushArrayLowerBound     % ... %T, arraydesc, lower
<               oOperandSwapLeftAndDest         % ... arraydesc, %T, lower
<               @EmitCmp                        %       cmpl   lower, %T
<               oOperandPushJumpCondition(iJge) % if subscript >= lower
<               oEmitConditionalForwardBranch   %       jge    fMM
<               oFixPushLastAddress
<               oOperandPop                     % ... arraydesc, %T, lower
<               oOperandPop                     % ... arraydesc, %T
<
<               @EmitSubscriptAbort             %       call   subscriptAbort
<               oFixForwardBranch               % fMM:
<               oFixPopAddress
<               oFixForwardBranch               % fNM:
<               oFixPopAddress
<       | *:
<       ]

<       % Normalize subscript                           % ... arraydesc, %T
<       oOperandSwap                                    % ... %T, arraydesc
<       @OperandPushArrayLowerBound                     % ... %T, arraydesc, lower
<       oOperandSwapLeftAndDest                         % ... arraydesc, %T, lower
<       @OperandSubtractPop                             %       subl lower, %T
<
<       % Scale subscript by string size (1024)         % ... arraydesc, %T
<       oOperandPushMode(mManifest)
<       oOperandSetLength(word)
< %%%%%%%% MJM 26Mar16: Note that shift left by 10 bits = 1024
<       oOperandSetValue(ten)                           % ... arraydesc, %T, 10
<       oEmitDouble(iShl)                               %       shll   $10, %T
<       oOperandPop                                     % ... arraydesc, %T
<
<       % Add normalized and scaled subscript to array address
<       oOperandSwap                                    % ... %T, arraydesc
<       [ oOperandChooseMode
<       | mStatic:
<               % Optimize by folding array offset into array descriptor address
<               oOperandPushMode(mManifest)
<               oOperandSetLength(word)
<               oOperandSetValue(eight)
<               oOperandAddManifestValues       % ... %T, arraydesc+8, 8
<               oOperandPop                     % ... %T, arraydesc+8
```

```
<             @OperandForceAddressIntoTemp      %      mov    $arraydesc+8, %T2
<                                               % ... %T, %T2
<             % Add array address to normalized and scaled subscript
<             oEmitDouble(iAdd)                 %      addl   %T2, %T
<             @OperandPopAndFreeTemp            % ... %T
<       | *:
<             % Can't optimize
<             @OperandForceAddressIntoTemp      % ... %T, %T2
<             oOperandPushMode(mManifest)
<             oOperandSetLength(word)
<             oOperandSetValue(eight)           % ... %T, %T2, 8
<             oEmitDouble(iAdd)                 %      addl   $8, %T2
<             oOperandPop                       % ... %T, %T2
<             oEmitDouble(iAdd)                 %      addl   %T2, %T
<             @OperandPopAndFreeTemp            % ... %T
<       ]

<       % Result element address is in %T
<       oOperandSetMode(mTempIndirect)                % ... (%T)
<       oOperandSetLength(word);
<

2154,2216d1804
< %%%%%%%% MJM 25Mar16: Used by OperandSubscriptStringPop
< %%%%%%%%                for tSubscriptString***
< %%%%%%%%                Note that this behaves just like
< %%%%%%%%                OperandSubscriptNonManifestIntegerPop
< %%%%%%%%                (i.e. tSubscriptInteger), but
< %%%%%%%%                with scaling factor of 1024 vs 4
< OperandUncheckedSubscriptNonManifestStringPop:
<       % Optimized non-bounds checking subscript operation
<       oOperandSwap                                  % ... subscript, arraydesc
<
<       [ oOperandChooseMode
<       | mTempIndirect:
<             % Var parameter array - don't know the characteristics
<             % until run time, so give up and use regular checked subscripting
<             oOperandSwap
<             @OperandCheckedSubscriptNonManifestStringPop
<
<       | mStatic:
<             % Any other array - know all the characteristics now,
<             % so optimize subscripting as best we can
<             oOperandSwap                      % ... arraydesc, subscript
<
<             % Scale subscript by string element size (1024)
<             @OperandForceIntoTemp             %      movl   subscript, %T
<             oOperandPushMode(mManifest)
<             oOperandSetLength(word)

< %%%%%%%% MJM 26Mar16: Note that shift left by 10 bits = 1024
<             oOperandSetValue(ten)
<             oEmitDouble(iShl)                 %      shl    $10, %T
<             oOperandPop
<             oOperandSwap                      % ... %T, arraydesc
<
<             % Fold lower bound into array address to avoid normalizing
<             % subscript at run time
<             oOperandPushArrayLowerBound       % ... %T, arraydesc, lower
<             oOperandSwap                      % ... %T, lower, arraydesc
<             oOperandSetMode(mManifest)        % (eliminate indirection)
<             oOperandPushMode(mManifest)       % ... %T, lower, arraydesc, 8
```

```
<             oOperandSetLength(word)
<             oOperandSetValue(eight)
<             oOperandAddManifestValues         % ... %T, lower, arraydesc+8, 8
<             oOperandPop                       % ... %T, lower, arraydesc+8
<             oOperandSwap                      % ... %T, arraydesc+8, lower
<             oOperandPushCopy                  % ... %T, arraydesc+8, lower, lower
<             oOperandAddManifestValues         % (scale lower bound by integer size)
<             oOperandPop
<             oOperandPushCopy
<             oOperandAddManifestValues         % ... %T, arraydesc+8, lower*4, lower
<             oOperandPop                       % ... %T, arraydesc+8, lower*4
<             oOperandSubtractManifestValues    % ... %T, arraydesc+8-lower*4, lower*4
<             oOperandPop                       % ... %T, arraydesc+8-lower*4
<
<             % Add array base to subscript
<             oOperandSetMode(mStaticManifest)  % (u+normalizedArrayBase)
<             oEmitDouble(iAdd)                 %      addl   $u+normalizedArrayBase,
%T
<             oOperandPop                       % ... %T
<
<             % Element address is in %T
<             oOperandSetMode(mTempIndirect)    % ... (%T)
<             oOperandSetLength(word)
<      ];
```

- FILE: "coder.pt" (new = “<”, old = “>”)

```
508d493
<      stringSize = 1024;         { MJM 2Apr16: size of string is 1024 bytes }

1826,1833c1793
<                           subscript := subscript * wordSize
< {%%%%%%%% MJM 2Apr16: Add case for string arrays of size stringSize
< %%%%%%%%}
<                     else if operandStkLength[operandStkTop-1] = string then
<                           { Convert a byte offset to a string offset }
<                           subscript := subscript * stringSize;
<
<
---
>                           subscript := subscript * wordSize;
```

3.7 Support Empty Strings

Changes in this section of “coder.pt” are required to support strings with zero length (i.e. blank).

- FILE: "coder.pt" (new = “<”, old = “>”)

```
980d964
<             {MPM 2th April: Change loop to handle empty string case ""}
985,999c969
<             {
<             MPM 2th April: Change loop to handle empty string case "".
<             If there is empty string, we output a whitespace instead.
```

```
<             }
<             if(compoundTokenLength = 0) then
<             begin
<             write(object, ' ');
<             end
<      else
<      {
<             Otherwise, handle it as usual, add one assertion that
<             compundTokenLength does not equal to zero.
<      }
<             Assert( (compoundTokenLength <> 0), assert5);
<      begin
---
>
1014c984
<      end;

1921,1926c1877
<                                  {MPM 2th April: add handle of empty string ""
<                                  if there is empty string "", we just ignore it
<                                  from the input side and handle it in
<                                  the EmitX86StringLiteral function instead.
<                                  }
<                                  Assert ( (compoundTokenLength >= 0), assert5);
---
>                                  Assert ( (compoundTokenLength >= 1), assert5);
1928,1930d1878
<                                  if(compoundTokenLength <> 0) then
<                                  begin
<
1938d1885
<
1940,1941c1887
<                                  compoundTokenText[compoundTokenLength + 1] := null
<                                  end
---
>                                  compoundTokenText[compoundTokenLength+1] := null
```

3.8 Code Templates for String Operations.

The following changes list the code templates added into the code generator to handle the new string operations. Most of these templates invoke PTTRAP functions found in the PT run-time library.

- FILE: "`coder.ssl`" (new = “<”, old = “>”)

```
1622,1675d1475
<
< ######## MJM 25Mar16: Added code templates for the following:
< ########                 1. tAssignString
< ########                 2. tConcatenate
< ########                 3. tSubstring
< ########                 4. tLength
```

```
< %%%%%%%%                5. tStringEqual
<       | tAssignString:
<             @OperandAssignStringPopPop
<       | tConcatenate:
<             @OperandConcatenatePop
<       | tSubstring:
<             @OperandSubstringPopPop
<       | tLength:
<             @OperandLength
<       | tStringEqual:
<             @OperandStringEqualPop
<
<

1776,1783d1564
<
< %%%%%%%% MJM 25Mar16: ***Code Template for tFetchString***
<       | tFetchString:
<             % MJM: Get the string address (s1) on Operand Stack
<             %      and put into temp register %T
<             @OperandForceAddressIntoTemp     %      lea    s1, %T
<
<

1811,1837d1591
< %%%%%%%% MJM 25Mar16: ***Code Template for tSubscriptString***
< %%%%%%%%                Note that this behaves just like
< %%%%%%%%                OperandSubscriptIntegerPop
< %%%%%%%%                (i.e. tSubscriptInteger), but
< %%%%%%%%                with scaling factor of 1024 vs 4
< OperandSubscriptStringPop:
<       % if the subscript is manifest fold it out,
<       % otherwise generate subscripting code
<       [ oOperandChooseMode
<       | mManifest:
<             oOperandSwap
<             [ oOperandChooseMode
<                   | mTempIndirect:
<                   % var parameter subscripting cannot be folded
<                   oOperandSwap
<                   @OperandSubscriptNonManifestStringPop
<             | *:
<                   oOperandSwap              % ... array, subscript
<                   oOperandFoldManifestSubscript
<                   oOperandPop               % ... array[subscript]
<             ]
<       | *:
<             @OperandSubscriptNonManifestStringPop
<       ]
<       oOperandSetLength(word);


< %%%%%%%% MJM 25Mar16: Added code templates for the following:
< %%%%%%%%                1. tAssignString
< %%%%%%%%                2. tConcatenate
< %%%%%%%%                3. tSubstring
< %%%%%%%%                4. tLength
< %%%%%%%%                5. tStringEqual
<       | tAssignString:
<             @OperandAssignStringPopPop
<             >
<       | tConcatenate:
```

```
<             @OperandConcatenatePop
<       | tSubstring:
<             @OperandSubstringPopPop
<       | tLength:
<             @OperandLength
<       | tStringEqual:
<             @OperandStringEqualPop
<
---
>       | tAssignChar:
>             @OperandAssignCharPopPop
>             >

2549,2833d2055
< %%%%%%%% MJM 25Mar16: 'tAssignString' ***Code Template implementation***
< %%%%%%%%                  This assumes: 's1' = target string variable
< %%%%%%%%                        and 's2' = asigned string variable
< %%%%%%%%
< OperandAssignStringPopPop:
<       @SaveTempRegsToStack              % pushl %eax .. %edx
<
<       % Operand Stack looks like this now  ... s1, s2
<
<       % Get string address (s2) on Operand Stack & put into %T
<       @OperandForceAddressIntoTemp      %      lea    s2, %T
<       @OperandForceToStack              %      pushl   %T
<       @OperandPopAndFreeTemp            %      ... s1
<
<       % Get string address (s1) on Operand Stack & put into %T
<       @OperandForceAddressIntoTemp      %      lea    s1, %T
<       @OperandForceToStack              %      pushl   %T
<       @OperandPopAndFreeTemp            %      ...
<
<       % Run-time Stack looks like this now ... s2, s1
<
<       % Setup Trap
<       oOperandPushMode(mTrap)
<       oOperandSetValue(pttrap101)
<       oOperandSetLength(word)
<
<       % Call Trap routine
<       oEmitSingle(iCall)                %      call   pttrap101
<       oOperandPop
<
<       % Pop arguments from Run-time Stack
<       oOperandPushMode(mStackReg)
<       oOperandSetLength(word)
<       oOperandPushMode(mManifest)
<       oOperandSetLength(word)
<       oOperandSetValue(eight)
<       oEmitDouble(iAdd)                 %      addl   $8, %esp
<       oOperandPop
<       oOperandPop
<       % Run-time Stack looks like this now  ...

<       % Restore temp regs
<       @RestoreTempRegsFromStack         %      popl   %edx .. $eax
< ;
<
< %%%%%%%% MJM 25Mar16: 'tConcatenate' ***Code Template implementation***
< %%%%%%%%                  This assumes: 's1' = string1 address
< %%%%%%%%                        and 's2' = string2 address
```

```
< %%%%%%%%                Resultant string address put into temporary Register %T
< %%%%%%%%
< OperandConcatenatePop:
<       @SaveTempRegsToStack              % pushl %eax .. %edx
<
<       % Operand Stack looks like this now  ... s1, s2
<
<       % Get string address (s2) on Operand Stack & put into %T
<       @OperandForceAddressIntoTemp      %       lea    s2, %T
<       @OperandForceToStack              %       pushl   %T
<       @OperandPopAndFreeTemp            %       ... s1
<
<       % Get string address (s1) on Operand Stack & put into %T
<       @OperandForceAddressIntoTemp      %       lea    s1, %T
<       @OperandForceToStack              %       pushl   %T
<       @OperandPopAndFreeTemp            %       ...
<
<       % Run-time Stack looks like this now ... s2, s1
<
<       % Setup Trap
<       oOperandPushMode(mTrap)
<       oOperandSetValue(pttrap103)
<       oOperandSetLength(word)
<
<       % Call Trap routine
<       oEmitSingle(iCall)                %       call   pttrap103
<       oOperandPop

<
<       % Pop arguments from Run-time Stack
<       oOperandPushMode(mStackReg)
<       oOperandSetLength(word)
<       oOperandPushMode(mManifest)
<       oOperandSetLength(word)
<       oOperandSetValue(eight)
<       oEmitDouble(iAdd)                 %       addl   $8, %esp
<       oOperandPop
<       oOperandPop
<       % Run-time Stack looks like this now  ...
<
<       % Accept result
<       oOperandPushMode(mScratchReg1)    %       ... %S1
<       oOperandSetLength(word)
<       oOperandPushMode(mResultReg)      %       ... %S1, %eax
<       oOperandSetLength(word)
<       oEmitDouble(iMov)                 %       movl   %eax, %S1
<       oOperandPop %eax                  %       ... %S1
<
<       % Restore temp regs
<       @RestoreTempRegsFromStack         %       popl   %edx .. $eax
<
<       % Put result back into a temp register within the previous scope
<       @OperandForceIntoTemp             %       movl %S1, %T
<
<       % Since this is an address result, set this to "word"
<       oOperandSetLength(word)
< ;
<
<


<
< %%%%%%%% MJM 25Mar16: 'tSubString' ***Code Template implementation***
```

```
< %%%%%%%%                  This assumes: 's1' = string address
< %%%%%%%%                        and 'i1' = lower index
< %%%%%%%%                        and 'i2' = upper index
< %%%%%%%%                  Result substring address put into temporary Register %T
< %%%%%%%%
< OperandSubStringPopPop:
<       @SaveTempRegsToStack              % pushl %eax .. %edx
<
<       % Operand Stack looks like this now  ... s1, i1, i2
<
<       % Get upper index (i2) on Operand Stack & push onto RT stack
<       @OperandForceToStack              %       pushl   i2
<       @OperandPopAndFreeTemp            %       ... s1, i1
<
<       % Get upper index (i1) on Operand Stack & push onto RT stack
<       @OperandForceToStack              %       pushl   i1
<       @OperandPopAndFreeTemp            %       ... s1
<
<       % Get string address (s1) on Operand Stack & put into %T
<       @OperandForceAddressIntoTemp      %       lea    s1, %T
<       @OperandForceToStack              %       pushl   s1
<       @OperandPopAndFreeTemp            %       ...
<
<       % Run-time Stack looks like this now ... i2, i1, s1
<
<       % Setup Trap
<       oOperandPushMode(mTrap)
<       oOperandSetValue(pttrap104)
<       oOperandSetLength(word)
<
<       % Call Trap routine
<       oEmitSingle(iCall)                %       call   pttrap104
<       oOperandPop

<       % Pop arguments from Run-time Stack
<       oOperandPushMode(mStackReg)
<       oOperandSetLength(word)
<       oOperandPushMode(mManifest)
<       oOperandSetLength(word)
<       oOperandSetValue(twelve)
<       oEmitDouble(iAdd)                 %       addl   $12, %esp
<       oOperandPop
<       oOperandPop
<       % Run-time Stack looks like this now  ...
<
<       % Accept result
<       oOperandPushMode(mScratchReg1)    %       ... %S1
<       oOperandSetLength(word)
<       oOperandPushMode(mResultReg)      %       ... %S1, %eax
<       oOperandSetLength(word)
<       oEmitDouble(iMov)                 %       movl   %eax, %S1
<       oOperandPop %eax                  %       ... %S1
<
<       % Restore temp regs
<       @RestoreTempRegsFromStack         %       popl   %edx .. $eax
<
<       % Put result back into a temp register within the previous scope
<       @OperandForceIntoTemp             %       movl %S1, %T
<
<       % Since this is an address result, set this to "word"
<       oOperandSetLength(word)
< ;
```

```
< %%%%%%%% MJM 25Mar16: 'tLength' ***Code Template implementation***
< %%%%%%%%                 This assumes: 's1' = string address
< %%%%%%%%                 Result integer length put into temporary Register %T
< %%%%%%%%
< OperandLength:
<      @SaveTempRegsToStack             % pushl %eax .. %edx
<
<      % Operand Stack looks like this now  ... s1
<
<      % Get string address (s1) on Operand Stack & put into %T
<      @OperandForceAddressIntoTemp     %      lea    s1, %T
<      @OperandForceToStack             %      pushl   s1
<      @OperandPopAndFreeTemp           %      ...
<
<      % Run-time Stack looks like this now ...  s1
<
<      % Setup Trap
<      oOperandPushMode(mTrap)
<      oOperandSetValue(pttrap105)
<      oOperandSetLength(word)
<
<      % Call Trap routine
<      oEmitSingle(iCall)               %      call   pttrap105
<      oOperandPop
<
<      % Pop arguments from Run-time Stack
<      oOperandPushMode(mStackReg)
<      oOperandSetLength(word)
<      oOperandPushMode(mManifest)
<      oOperandSetLength(word)
<      oOperandSetValue(four)
<      oEmitDouble(iAdd)                %      addl   $4, %esp
<      oOperandPop
<      oOperandPop
<      % Run-time Stack looks like this now  ...
<
<      % Accept result
<      oOperandPushMode(mScratchReg1)   %      ... %S1
<      oOperandSetLength(word)
<      oOperandPushMode(mResultReg)     %      ... %S1, %eax
<      oOperandSetLength(word)
<      oEmitDouble(iMov)                %      movl   %eax, %S1
<      oOperandPop %eax

<      % Restore temp regs
<      @RestoreTempRegsFromStack        %      popl   %edx .. $eax
<
<      % Put result back into a temp register within the previous scope
<      @OperandForceIntoTemp            %      movl %S1, %T
<
<      % Since this is an integer result, set this to "word"
<      oOperandSetLength(word)
< ;

< %%%%%%%% MJM 25Mar16: 'tStringEqual' ***Code Template implementation***
< %%%%%%%%                 This assumes: 's1' = string1 address
< %%%%%%%%                       and 's2' = string2 address
< %%%%%%%%                 Result boolean put into temporary Register %T
< %%%%%%%%
< OperandStringEqualPop:
<      @SaveTempRegsToStack             % pushl %eax .. %edx
```

```
<
<       % Operand Stack looks like this now  ... s1, s2
<
<       % Get string address (s2) on Operand Stack & put into %T
<       @OperandForceAddressIntoTemp     %      lea    s2, %T
<       @OperandForceToStack             %      pushl   %T
<       @OperandPopAndFreeTemp           %      ... s1
<
<       % Get string address (s1) on Operand Stack & put into %T
<       @OperandForceAddressIntoTemp     %      lea    s1, %T
<       @OperandForceToStack             %      pushl   %T
<       @OperandPopAndFreeTemp           %      ...
<
<       % Run-time Stack looks like this now ... s2, s1
<
<       % Setup Trap
<       oOperandPushMode(mTrap)
<       oOperandSetValue(pttrap106)
<       oOperandSetLength(word)
<
<       % Call Trap routine
<       oEmitSingle(iCall)               %      call   pttrap106
<       oOperandPop

<       % Pop arguments from Run-time Stack
<       oOperandPushMode(mStackReg)
<       oOperandSetLength(word)
<       oOperandPushMode(mManifest)
<       oOperandSetLength(word)
<       oOperandSetValue(eight)
<       oEmitDouble(iAdd)                %      addl   $8, %esp
<       oOperandPop
<       oOperandPop
<       % Run-time Stack looks like this now  ...
<
<       % Accept result
<       oOperandPushMode(mScratchReg1)   %      ... %S1
<       oOperandSetLength(word)
<       oOperandPushMode(mResultReg)     %      ... %S1, %eax
<       oOperandSetLength(word)
<       oEmitDouble(iMov)                %      movl   %eax, %S1
<       oOperandPop %eax                 %      ... %S1
<
<       % Restore temp regs
<       @RestoreTempRegsFromStack        %      popl   %edx .. $eax
<
<       % Put result back into a temp register within the previous scope
<       @OperandForceIntoTemp            %      movl %S1, %T
<
<       % Since this is a boolena result, set this to "byte"
<       oOperandSetLength(byte)
< ;


<
< %%%%%%%% MJM 25Mar16: 'tChr' ***Code Template implementation***
< %%%%%%%%                 This assumes: 'i1' = integer value
< %%%%%%%%                 Resultant string address put into temporary Register %T
< %%%%%%%%
< OperandChr:
---
> OperandChrAssignPopPop:
```

```
3378d2582
<      % Run-time Stack looks like this now ... i2, i1, s1
3380,3383c2584,2587
<      % Setup Trap
<      oOperandPushMode(mTrap)
<      oOperandSetValue(pttrap102)
<      oOperandSetLength(word)
---
> OperandChr:
>      % Assume operand's value is in range
>      @OperandForceIntoTemp
>      oOperandSetLength(byte);  % see comments above
3385,3387d2588
<      % Call Trap routine
<      oEmitSingle(iCall)               %      call   pttrap102
<      oOperandPop
3389,3391c2590,2591
<      % Pop arguments from Run-time Stack
<      oOperandPushMode(mStackReg)
<      oOperandSetLength(word)
---
> OperandOrdAssignPopPop:
>      % x := y;  x word, y byte
3392a2593
>      oOperandSetValue(zero)
3394,3403c2595,2599
<      oOperandSetValue(four)
<      oEmitDouble(iAdd)                %      addl   $4, %esp
<      oOperandPop
<      oOperandPop
<      % Run-time Stack looks like this now  ...
<
<      % Accept result
<      oOperandPushMode(mScratchReg1)   %      ... %S1
<      oOperandSetLength(word)
<      oOperandPushMode(mResultReg)     %      ... %S1, %eax
---
>      @OperandForceIntoTemp      %      movl   $0, %T
>      oOperandSetLength(byte)
>      oOperandSwap
>      oEmitDouble(iMov)          %      movb   y, %T
>      @OperandPopAndFreeTemp

3405,3412c2601,2603
<      oEmitDouble(iMov)                %      movl   %eax, %S1
<      oOperandPop %eax                 %      ... %S1
<
<      % Restore temp regs
<      @RestoreTempRegsFromStack        %      popl   %edx .. $eax
<
<      % Put result back into a temp register within the previous scope
<      @OperandForceIntoTemp            %      movl %S1, %T
---
>      oEmitDouble(iMov)                %      movl   %T, x
>      @OperandPopAndFreeTemp
>      @OperandPopAndFreeTemp;
3414,3416d2604
<      % Since this is an address result, set this to "word"
<      oOperandSetLength(word)
< ;
3418,3439d2605
```

```
< %%%%%%%% MJM 25Mar16: OperandOrdAssignPopPop is no longer used; this
< %%%%%%%%                optimization doesn't make sense for strings.
< %OperandOrdAssignPopPop:
< %    % x := y;  x word, y byte
< %    oOperandPushMode(mManifest)
< %    oOperandSetValue(zero)
< %    oOperandSetLength(word)
< %    @OperandForceIntoTemp      %      movl   $0, %T
< %    oOperandSetLength(byte)
< %    oOperandSwap
< %    oEmitDouble(iMov)                 %      movb   y, %T
< %    @OperandPopAndFreeTemp
< %    oOperandSetLength(word)
< %    oEmitDouble(iMov)                 %      movl   %T, x
< %    @OperandPopAndFreeTemp
< %    @OperandPopAndFreeTemp;
<

< %%%%%%%% MJM 25Mar16: 'tChr' ***Code Template implementation***
< %%%%%%%%                This assumes: 's1' = string address
< %%%%%%%%                Result integer put into temporary Register %X
< %%%%%%%%
3441,3445c2607
<      % Operand Stack looks like this now  ... s1
<
<      % Get string address (s1) on Operand Stack & put into %T
<      @OperandForceAddressIntoTemp      %      lea    s1, %T
<                                        % ... %T
---
>      % Byte operand is on top of operand stack
3448,3453c2610,2611
<      oOperandSetLength(word)           % ... %T, $0
<      @OperandForceIntoTemp             %      movl   $0, %X
<                                        % ... %T, %X
<
<      oOperandSwap                      % ... %X, %T
<      oOperandSetMode(mTempIndirect)    % ... %X, (%T)
---
>      oOperandSetLength(word)
>      @OperandForceIntoTemp      %      movl   $0, %T
3455,3457c2613,2615
<      oEmitDouble(iMov)                 %      movb (%T), %X
<      @OperandPopAndFreeTemp            % ... %X and free temp
<
---
>      oOperandSwap
>      oEmitDouble(iMov)          %      movb   y, %T
>      @OperandPopAndFreeTemp
< %%%%%%%% MJM 25Mar16: Use the 'address' version of this routine
< %%%%%%%%                if OperandMode 'mString'
<      | mString:
<            @OperandForceAddressIntoTemp
```

3.9 Add Handling of Loop Statement & Remove While, Repeat/Until Statements.

Changes here involve the removal of handling of Repeat/Until loops and While loops.

- FILE: "coder.ssl" (new = “<”, old = “>”)

```
946,955c844,847
< %%%%%%%% MJM 25Mar16: Change from While to Loop structure
< %%%%%%%%                 adn renamed appropriate rules and t-codes.
<       | tLoopBegin:
<             @LoopStmt
<
< %%%%%%%% MJM 25Mar16: Repeat structure not used any more.
< %     | tRepeatBegin:
< %           @RepeatStmt
< %%%%%%%%
<
---
>       | tWhileBegin:
>             @WhileStmt
>       | tRepeatBegin:
>             @RepeatStmt
1133,1137c1025
< %%%%%%%% MJM 25Mar16: Change from While to Loop structure
< %%%%%%%%                 which involves adding optional statements
< %%%%%%%%                 before and after the condition check.
< %%%%%%%%                 Also, renamed from WhileStmt to LoopStmt
< LoopStmt:
---
> WhileStmt:
1141,1149d1028
<
< %%%%%%%% MJM 25Mar16: Note that this block of statements
< %%%%%%%%                 before the test condition is optional
<       @Statements                                     % loop body
<
< %%%%%%%% MJM 25Mar16: do statements then check for condition
< %%%%%%%%                 Note that Loop conditions are prefaced by
< %%%%%%%%                 tLoopBreakWhen, and followed by tLoopTest
<       tLoopBreakWhen
1150a1030,1031
>       tWhileTest
1141,1149d1028
<
< %%%%%%%% MJM 25Mar16: Note that this block of statements
< %%%%%%%%                 before the test condition is optional
<       @Statements                                     % loop body
<
< %%%%%%%% MJM 25Mar16: do statements then check for condition
< %%%%%%%%                 Note that Loop conditions are prefaced by
< %%%%%%%%                 tLoopBreakWhen, and followed by tLoopTest
<       tLoopBreakWhen
1150a1030,1031
>       tWhileTest
>       oOperandComplementJumpCondition         % ... !cond
1152d1032
<       tLoopTest % MJM 25Mar16: Change from tWhileTest to tLoopTest
1156c1036
<             % Exit condition is always false (break when false) - an infinite loop.
---
>             % Exit condition is always false (while true) - an infinite loop.
1161c1041
<             % Exit condition is always true (break when true) - a nop.
---
>             % Exit condition is always true (while false) - a nop.
1164c1044
```

```
<            @SkipToEndLoop % MJM 25Mar16: Change from 'While' to 'Loop'
---
>            @SkipToEndWhile
1174c1054
<            @OperandInfixOr                  %      jcond   fNNN
---
>            @OperandInfixOr                  %      j!cond  fNNN
1177,1179d1056
<
< %%%%%%%% MJM 25Mar16: Note that this block of statements
< %%%%%%%%                after the test condition is optional
1181,1182c1058
<
<      tLoopEnd % MJM 25Mar16: Change from tWhileEnd to tLoopEnd
---
>      tWhileEnd

1194,1197c1069,1070
<      | tLoopBegin: % MJM 25Mar16: Change from tWhileBegin to tLoopBegin
<                           % ignore nested while statements
<            @SkipToEndLoop % MJM 25Mar16: Change from SkipToEndWhile
<                           %                        to SkipToEndLoop
---
>      | tWhileBegin:                         % ignore nested while statements
>            @SkipToEndWhile
1202,1236c1075,1109
< %%%%%%%% MJM 25Mar16: Repeat structure not used any more.
< %RepeatStmt:
< %    % Save the target address for the top-of-loop branch
< %    oFixPushAddress                        % bNNN:
< %    oEmitMergeSourceCoordinate
< %    @Statements                                  % loop body
< %    tRepeatControl
< %    @OperandPushBooleanControlExpression    % ... cond
< %    tRepeatTest
< %
< %    % Optimize if loop condition is known at compile time
< %    [ oOperandChooseJumpCondition
< %    | iJnever:
< %          % Exit condition is always false (until false) - an infinite loop.
< %          % false branches fall through to top-of-loop backward branch
< %          oFixAndFreeFalseBranches
< %          oEmitUnconditionalBackwardBranch        %  jmp       bNN
< %
< %          | iJalways:
< %          % Exit condition is always true (until true) - loop never repeats.
< %          % fall through without backward branch
< %          oFixAndFreeFalseBranches
< %
< %    | *:
< %          % Emit a conditional forward branch to exit the loop.
< %          % True branches follow the conditional exit path, false
< %          % branches fall through to the top-of-loop branch.
< %          @OperandInfixOr                  %      jcond   fNNN
< %          oEmitUnconditionalBackwardBranch %      jmp    bNNN
< %    ]
< %
< %    oFixPopAddress
< %    % Fix the true branches exiting the statement
< %    oFixAndFreeShuntList                   % fNNN:
< %    oOperandPop;
---
```

```
>
> RepeatStmt:
>       % Save the target address for the top-of-loop branch
>       oFixPushAddress                         % bNNN:
>       oEmitMergeSourceCoordinate
>       @Statements                                     % loop body
>       tRepeatControl
>       @OperandPushBooleanControlExpression    % ... condl
>       tRepeatTest
>
>       % Optimize if loop condition is known at compile time
>       [ oOperandChooseJumpCondition
>       | iJnever:
>             % Exit condition is always false (until false) - an infinite loop.
>             % false branches fall through to top-of-loop backward branch
>             oFixAndFreeFalseBranches
>             oEmitUnconditionalBackwardBranch  %  jmp       bNN
>
>       | iJalways:
>             % Exit condition is always true (until true) - loop never repeats.
>             % fall through without backward branch
>             oFixAndFreeFalseBranches
>
>       | *:
>             % Emit a conditional forward branch to exit the loop.
>             % True branches follow the conditional exit path, false
>             % branches fall through to the top-of-loop branch.
>             @OperandInfixOr                 %      jcond   fNNN
>             oEmitUnconditionalBackwardBranch %     jmp    bNNN
>       ]
>
>       oFixPopAddress
>       % Fix the true branches exiting the statement
>       oFixAndFreeShuntList                    % fNNN:
>       oOperandPop;
```

3.10 Handle Function Return Type Declarations.

Changes here are required to support user-defined functions; specifically handling return values (i.e. within the Function declaration).

- FILE: "`coder.ssl`" (new = “<”, old = “>”)

```
899,919d817
<
< %%%%%%%% MJM 21Mar16: handle case of tFunctionResult, and take
< %%%%%%%%                 Return expression, evaluate, then put
< %%%%%%%%                 into the result register (mResultReg)
<       [
<       | tFunctionResult:
<             @OperandPushExpression     % ... x
<
<             %% Pop from Operand Stack and put into eax
<             oOperandPushMode(mResultReg)  % ... x, %eax
<             oOperandSetLength(word)
<             oEmitDouble(iMov)          %      movl   x, %eax
```

```
<             oOperandPop %eax           % ... x
< %%%%% Note that we use OperandPopAndFreeTemp to clean up temp registers
<             @OperandPopAndFreeTemp %x  % ...
< %           oOperandPop         %x     % ...
<
<
<       | *: % therefore not a function
<       ]
<
```

3.11 Add Handling Function Calls to User-Defined Functions

Changes here are required to support statements and expressions that call user-defined functions, and how to retrieve return values from those functions.

- FILE: "`coder.ssl`" (new = “<”, old = “>”)

```
2413,2440d1956
<
< %%%%%%%% MJM 21Mar16: handle function calls within expressions by
< %%%%%%%%                calling function, then retrieving result
< %%%%%%%%                from the result register (mResultReg)
<       | tCallBegin:
<
<             @SaveTempRegsToStack             % pushl %eax .. %edx
<
<             @CallStmt
<             %% Note that eax contains result for a function call
<
<             % Accept result from eax
<             oOperandPushMode(mScratchReg1)   % ... %S1
<             oOperandSetLength(word)
<             oOperandPushMode(mResultReg)     % ... %S1, %eax
<             oOperandSetLength(word)
<
<             % move result into scratchpad
<             oEmitDouble(iMov)                % movl %eax, %S1
<             oOperandPop %eax                 % ... %S1
<
<             % Restore temp regs
<             @RestoreTempRegsFromStack        % popl %edx .. $eax
<
<             @OperandForceIntoTemp            % movl %S1, %T
<             oOperandSetLength(word)
<
```