Phase 4. Code Generation

The Compildres

Harold Treen
Paul Wells

April 6th, 2015

1 - Updating Tokens

As with all previous phases, the input and output tokens had to be updated in coder.ssl. Tokens were added for the new string operations and loop syntax. Old tokens for char operations were removed.

```
85      % Added non-compound T-Codes
86      tFetchString
87      tAssignString
88      tStoreParmString
89      tSubscriptString
90      tConcatenate
91      tSubstring
92      tLength
93      tStringEqual
94      tLoopBegin
95      tLoopBreakIf
96      tCaseDefault
97      tCaseElse
98      % Compound T-codes are those that take operands
99      tLiteralAddress
100     firstCompoundToken = tLiteralAddress
```

```
116     tLineNumber
117     % Added Compound T-Codes
118     tLoopTest
119     tLoopEnd
120     %%
121     tTrap
```

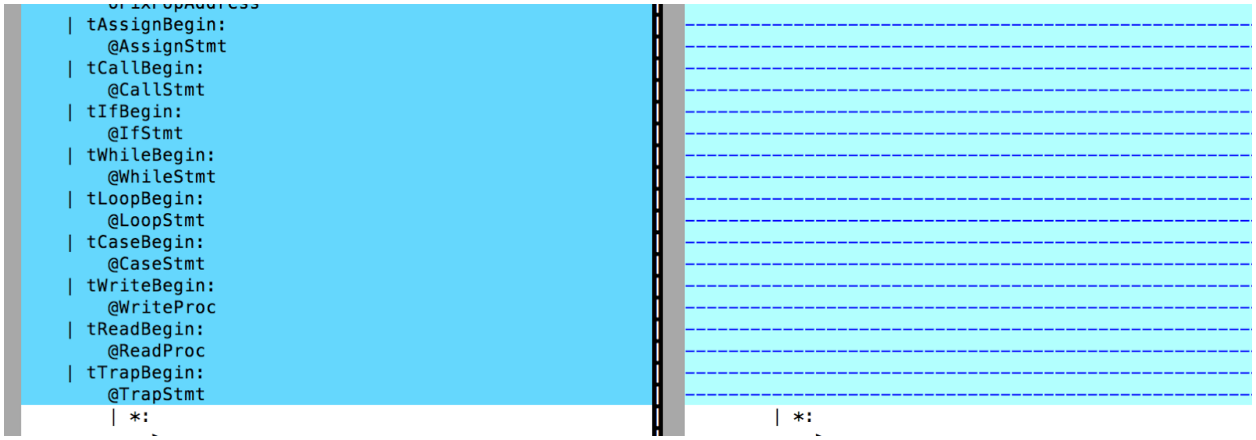
```
288     % Return the top entry's jump condition (kind field)
289
290     oOperandPushString
291     % Push an entry with addressing mode manifest, length w
292     % a value set to the data area address of the string re
```

```
56      tFetchInteger
57      tFetchChar
58      tFetchBoolean
59      tAssignBegin
60      tAssignAddress
61      tAssignInteger
62      tAssignChar
63      tAssignBoolean
64      tStoreParmAddress
65      tStoreParmInteger
66      tStoreParmChar
67      tStoreParmBoolean
68      tSubscriptBegin
69      tSubscriptAddress
70      tSubscriptInteger
71      tSubscriptChar
72      tSubscriptBoolean
73      tArrayDescriptor
74      tFileDescriptor
75      tIfBegin
76      tIfEnd
77      tCaseBegin
78      tWhileBegin
79      tRepeatBegin
80      tRepeatControl
81      tCallBegin
82      tParmEnd
83      tProcedureEnd
84      tWriteBegin
85      tReadBegin
86      tTrapBegin
87      tWriteEnd
88      tReadEnd
89
90      % Compound T-codes are those that take operands
91      tLiteralAddress
92      firstCompoundToken = tLiteralAddress
93      tLiteralInteger
94      tLiteralChar
95      tLiteralBoolean
```

Once the tokens in the .ssl file were updated, a .def file was generated which was used to update definitions in coder.pt.

2 - Drift Modules

Since modules are essentially a scoping construct, most of the work for modules has already been done with one exception: Now, Statements do not strictly have to follow Declarations. Therefore in the Block rule we added the choice alternatives for statements into the loop with declarations.



3 - Drift Loop Templates

For Drift loops the first step was to remove the old RepeatStmt Rule. The RepeatStmt rule was replaced with a LoopStmt rule. As you can see below, the diffs for this piece of code are messy.

```
    % Added Statements before break condition
    @Statements
    % Require break if
    tLoopBreakIF
    @OperandPushBooleanControlExpression    % ... cond
    % Changed from tWhileTest to tLoopTest
    tLoopTest
    %oOperandComplementJumpCondition     % ... !cond
```

```
RepeatStmt:
    % Save the target address for the top-of-loop branch
    oFixPushAddress                                % bNNN:
    oEmitMergeSourceCoordinate
    @Statements                                    % loop bod
    tRepeatControl
    @OperandPushBooleanControlExpression           % ... cond
    tRepeatTest

    % Optimize if loop condition is known at compile time
    [ oOperandChooseJumpCondition
          | iJnever:
              % Exit condition is always false (until false)
              % false branches fall through to top-of-loop b
              oFixAndFreeFalseBranches
              oEmitUnconditionalBackwardBranch     %  jmp

      | iJalways:
          % Exit condition is always true (until true) -
          % fall through without backward branch
          oFixAndFreeFalseBranches
```

You can see that the RepeatStmt was removed but to show that the LoopStmt rule has been added we'll just show code snippets of our current code instead of a diff for clarity.

```
LoopStmt:
    % Save the target address for the top-of-loop branch
    oFixPushAddress             % bNNN:
    oEmitMergeSourceCoordinate    % accept and ignor

    % Added Statements before break condition
    @Statements

    % Require break if
    tLoopBreakIF
    @OperandPushBooleanControlExpression    % ... cond
    % Changed from tWhileTest to tLoopTest
    tLoopTest                                % loop bod
    %oOperandComplementJumpCondition     % ... !cond
    % Optimize if the condition is know at compile time
    [ oOperandChooseJumpCondition
    | iJnever:
        % Exit condition is always false (while true) - an infinite loop.
        % emit no conditional branch, just fall into loop body
        oFixAndFreeFalseBranches
    | iJalways:
        % Exit condition is always true (while false) - a nop.
        % flush the loop body
        oFixAndFreeFalseBranches
        @SkipToEndWhile
        oFixPopAddress
        oFixAndFreeShuntList            % fNN:, just in case
        oOperandPop
        >>

    | *:
        % Emit a conditional forward branch to exit the loop.
        % True (inverted false) branches follow the conditional
        % exit path, false branches fall through to the loop body.
        @OperandInfixOr          %   j!cond  fNNN
    ]

    @Statements                 % loop body

    % Changed from tWhileEnd to tLoopEnd
    tLoopEnd

    oEmitUnconditionalBackwardBranch        %   jmp bNNN
```

```
    oFixPopAddress
    oFixAndFreeShuntList            % fNNN:
    oOperandPop;
```

You can see that the code for a loop statement is similar to the code for a while loop except that statements are added before the break condition and of course tWhileEnd is changed to tLoopEnd.

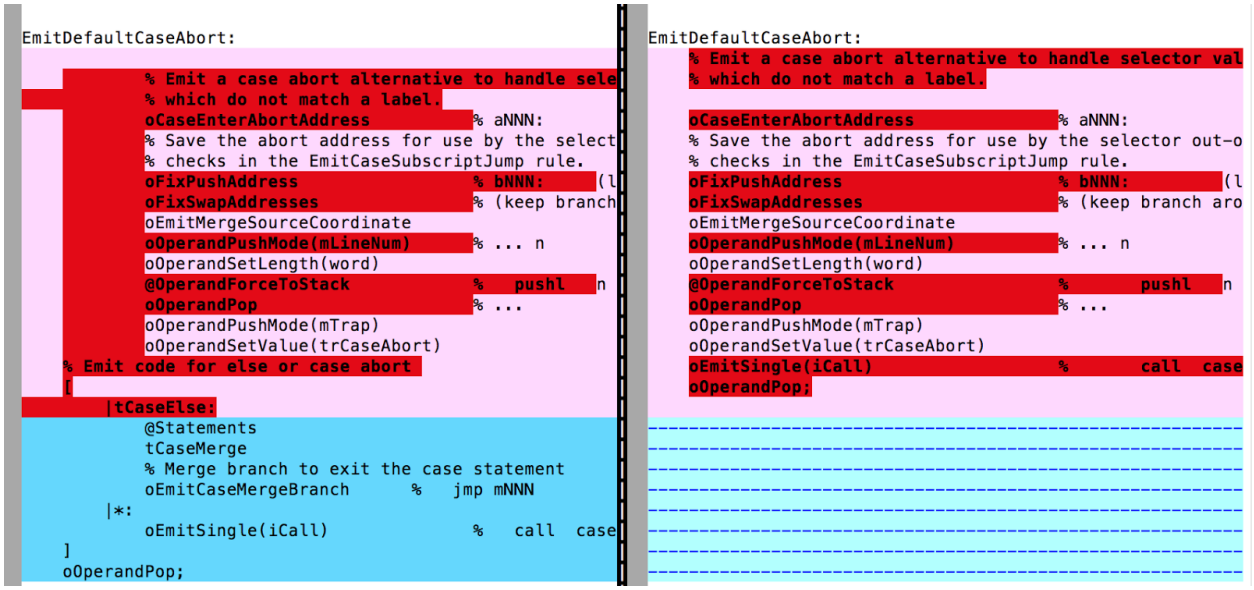
Also, the choice for tRepeatBegin had to be changed to tLoopBegin in the Block and Statement rules like so:

Finally, in coder.pt, tRepeatTest had to be replaced with tLoopTest and tLoopEnd inside the code which accepts compound tokens.

```
1868  1867          case compoundToken of
1869  1868              tLiteralAddress, tLiteralInteger,
1870        -         -     tLiteralBoolean,
1871        -         -     tWhileTest, tWhileEnd, tRepeatTest, tCallEnd,
      1869  +             tLiteralBoolean, tLoopTest, tLoopEnd,
      1870  +             tWhileTest, tWhileEnd, tCallEnd,
```

4 - Drift Switch Template

The only change that was required for drift switch statements was to edit the EmitDefaultCaseAbort rule so that if there is a tCaseElse token (indicating that there is a default case) then to execute that rule when no case matches are found instead of aborting the case statement.



5 - String Templates

5.1 - Literal Strings

tLiteralString allows string variables to be declared within the machine code. It replaced the tStringDescriptor cases from the old code generator. The choice was added to both the regualr “OperandPushExpression” and optimized “OperandPushExpressionAssignPopPop” rule.

```
1506      % Added choice for tLiteralString
1507      | tLiteralString:
1508          % Logic is the same except no tStringDescriptor acc
1509          % Emit string literal to data area
1510          oEmitNone(iData)              %    .data
1511          % Emit the string
1512          oEmitString               % sNNN: .asciz  "SSSSS"
1513          oEmitNone(iText)              %    .text
1514          % Get the string literal's address
1515          oOperandPushString
1516          @EmitStringDescriptor         %    lea sNNN, %T
```

coder.ssl

5.2 - Fetching String

Fetching strings was done by forcing the address of the string operand into a temp variable using the “EmitStringDescriptor” rule. It replaced the tFetchChar operation.

```
| tFetchChar, tFetchBoolean:
    oOperandSetLength(byte)
| *:
    % Value is not to be loaded
] ;
```

```
1614      | tFetchBoolean:
1615          oOperandSetLength(byte)
1616      | tFetchString:
1617          @EmitStringDescriptor
1618      | *:
1619          % Value is not to be loaded
```

coder.ssl

5.3 - Assign Strings

String assignment was added as a case in “OperandPushExpressionAssignPopPop”. The new rule saves all temporary registers, puts the string addresses in registers and does a trap call to move the contents at the first address to the address of the second. Both operands are then popped from the stack.

```
2182      % Add tAssignString
2183      | tAssignString:
2184          @OperandAssignStringPopPop
2185          >
```

```
3132  OperandAssignStringPopPop:
3133      % Generate code to assign the right (top) operand's val
3134      % the left (second) operand and pop both operands.  For
3135      @SaveTempRegsToStack
3136      @EmitStringDescriptor %lea s2, %T
3137      @OperandForceToStack  %pushl %T
3138      @OperandPopAndFreeTemp
3139      @EmitStringDescriptor %lea s1,  %T
3140      @OperandForceToStack %pushl, %T
3141      @OperandPopAndFreeTemp
3142
3143      % Make trap call
3144      oOperandPushMode(mTrap)
3145      oOperandSetValue(trAssignString)
3146      oOperandSetLength(word)
3147      oEmitSingle(iCall)
3148      oOperandPop
3149
3150      % Pop arguments
3151      oOperandPushMode(mStackReg)
3152      oOperandSetLength(word)
3153      oOperandPushMode(mManifest)
3154      oOperandSetLength(word)
3155      oOperandSetValue(eight)
3156      oEmitDouble(iAdd)
3157      oOperandPop
3158      oOperandPop
3159
3160      @RestoreTempRegsFromStack;
3161
```

coder.ssl

5.4 - Parameter Strings

String parameter handling was added so that strings could be passed to functions. The added code moves the string value into a variable which is accessible by the function.

```
  %removed tStoreParmChar from choice above and added tSt
 | tStoreParmString:
     oOperandSwap      % ... formal, 8(%ebp)
     oOperandPushCopy      % ... formal, 8(%ebp), 8(%ebp)
     oOperandSwapLeftAndDest  % ... 8(%ebp), formal, 8(%eb
     oOperandSetLength(string)
     @OperandAssignStringPopPop
```

coder.ssl

5.5 - String Subscripting

String subscripting was done identically to integer subscripting. The difference was a small change so that the calculated offsets would be multiplied by 256 (the size of strings) by multiplying in coder.pt and shifting in coder.ssl.

```
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
        | *:
        oOperandSetLength(string)
        oOperandSwap        % ... array, subscript
            oOperandFoldManifestSubscript
            oOperandPop         % ... array[subscript]
        ]
    | *:
        @OperandSubscriptNonManifestStringPop
    ]
    oOperandSetLength(word);
```

```
1701 OperandSubscriptNonManifestStringPop:
1702     [ oOptionTestChecking
1703     | yes:
1704         @OperandCheckedSubscriptNonManifestStringPop
1705     | *:
1706         @OperandUncheckedSubscriptNonManifestStringPop
1707     ];
1708
```

```
1967 OperandUncheckedSubscriptNonManifestStringPop:
1968     % Optimized non-bounds checking subscript operation
1969     oOperandSwap                % ... subscript, arraydesc
1970
1971     [ oOperandChooseMode
1972     | mTempIndirect:
1973         % Var parameter array - don't know the characteristics
1974         % until run time, so give up and use regular checked subscripting
1975         oOperandSwap
1976         @OperandCheckedSubscriptNonManifestStringPop
1977
1978     | mStatic:
1979         % Any other array - know all the characteristics now,
1980         % so optimize subscripting as best we can
1981         oOperandSwap            % ... arraydesc, subscript
1982
1983         % Scale subscript by string element size
1984         @OperandForceIntoTemp       %   movl    subscript, %T
1985         oOperandPushMode(mManifest)
1986         oOperandSetLength(word)
1987         oOperandSetValue(eight)
1988         oEmitDouble(iShl)           %   shl     $2, %T
1989         oOperandPop
1990         oOperandSwap            % ... %T, arraydesc
1991
1992         % Fold lower bound into array address to avoid normalizing
1993         % subscript at run time
1994         oOperandPushArrayLowerBound     % ... %T, arraydesc, lower
1995         oOperandSwap            % ... %T, lower, arraydesc
1996         oOperandSetMode(mManifest)      % (eliminate indirection)
1997         oOperandPushMode(mManifest)     % ... %T, lower, arraydesc, 8
1998         oOperandSetLength(word)
1999         oOperandSetValue(eight)
2000         oOperandAddManifestValues       % ... %T, lower, arraydesc+8, 8
2001         oOperandPop             % ... %T, lower, arraydesc+8
2002         oOperandSwap            % ... %T, arraydesc+8, lower
2003         oOperandPushCopy            % ... %T, arraydesc+8, lower, lower
2004         oOperandAddManifestValues       % (scale lower bound by integer size
2005         oOperandPop
2006         oOperandPushCopy
2007         oOperandAddManifestValues       % ... %T, arraydesc+8, lower*4, lowe
2008         oOperandPop             % ... %T, arraydesc+8, lower*4
2009         oOperandSubtractManifestValues      % ... %T, arraydesc+8-lower*4, 1
2010         oOperandPop             % ... %T, arraydesc+8-lower*4
2011
2012         % Add array base to subscript
2013         oOperandSetMode(mStaticManifest)    % (u+normalizedArrayBase)
2014         oEmitDouble(iAdd)           %   addl    $u+normalizedArrayBase, %T
2015         oOperandPop             % ... %T
2016
2017         % Element address is in %T
2018         oOperandSetMode(mTempIndirect)      % ... (%T)
2019         oOperandSetLength(string)
2020     ];
2021
```

coder.ssl

```
506         wordSize = 4;          { longword (4 bytes) on SUN }
507         stringSize = 256;
508
```

```
1805            if operandStkLength[operandStkTop-1] = word then subscript := subscript * wordSize
1806                    { Convert a byte offset to a word offset }
1807
1808            else
1809            begin
1810                if operandStkLength[operandStkTop-1] = string then subscript := subscript * stringSize
1811                    { Convert a byte offset to a string offset}
1812
1813
1814            end;
```

coder.pt

5.6 - Concatenation Operation

A case for the concatenation operation was added to both the “OperandPushExpression” and “OperandPushExpressionAssignPopPop” rules. The new operation pushes two string addresses into temp registers which can then be accessed by the “trConcatenate” trap call. This operation returns the result in a temp register which is then saved and restored.

```
tConcatenate:
  @OperandConcatenatePop
tLength:
```

```
2355 OperandConcatenatePop:
2356     @SaveTempRegsToStack
2357     @EmitStringDescriptor %lea s1,  %T
2358     @OperandForceToStack
2359     @OperandPopAndFreeTemp
2360     @EmitStringDescriptor %lea s1,  %T
2361     @OperandForceToStack %pushl, %T
2362     @OperandPopAndFreeTemp
2363
2364     % Make trap call
2365     oOperandPushMode(mTrap)
2366     oOperandSetValue(trConcatenate)
2367     oOperandSetLength(word)
2368     oEmitSingle(iCall)  % call pttrap103
2369     oOperandPop
2370
2371     % Pop arguments
2372     oOperandPushMode(mStackReg)
2373     oOperandSetLength(word)
2374     oOperandPushMode(mManifest)
2375     oOperandSetLength(word)
2376     oOperandSetValue(eight)
2377     oEmitDouble(iAdd)
2378     oOperandPop
2379     oOperandPop
2380
2381     % Save result
2382     @SaveResult
2383
2384     @RestoreTempRegsFromStack
2385
2386     % Move result to temp
2387     @ResultToTemp;
2388
```

coder.ssl

5.7 - Length Operation

The length operation was also added as a case to both the “OperandPushExpression” and “OperandPushExpressionAssignPopPop” rules. The length operation pushes a string operand address into a temp register and then calls a trap which returns the length. The result is then saved and restored.

```
| tLength:
    @OperandLength
ubtract:
```

```
2389 OperandLength:
2390     @SaveTempRegsToStack
2391     @EmitStringDescriptor %lea s1, %T
2392     @OperandForceToStack  %pushl %T
2393     @OperandPopAndFreeTemp
2394
2395     % Make trap call
2396     oOperandPushMode(mTrap)
2397     oOperandSetValue(trLength)
2398     oOperandSetLength(word)
2399     oEmitSingle(iCall)  % call pttrap105
2400     oOperandPop
2401
2402     % Pop Arguments
2403     oOperandPushMode(mStackReg)
2404     oOperandSetLength(word)
2405     oOperandPushMode(mManifest)
2406     oOperandSetLength(word)
2407     oOperandSetValue(four)
2408     oEmitDouble(iAdd)
2409     oOperandPop
2410     oOperandPop
2411
2412     % Save result
2413     @SaveResult % movl eax , esi
2414
2415     @RestoreTempRegsFromStack
2416
2417     % Move result to temp
2418     @ResultToTemp;
2419
```

coder.ssl

5.8 - Chr Operation

The “OperandChr” rule was updated to work with the new string type. The code now saves the integer parameter in a register before calling a trap to return the corresponding string. The result is then saved and restored.

```
2851 OperandChr:
2852     @SaveTempRegsToStack
2853
2854     @OperandForceToStack % pushl i1
2855     @OperandPopAndFreeTemp
2856
2857     % Make trap call
2858     oOperandPushMode(mTrap)
2859     oOperandSetValue(trChrString) % call pttrap102
2860     oOperandSetLength(word)
2861     oEmitSingle(iCall)
2862     oOperandPop
2863
2864     % Pop arguments
2865     oOperandPushMode(mStackReg)
2866     oOperandSetLength(word)
2867     oOperandPushMode(mManifest)
2868     oOperandSetLength(word)
2869     oOperandSetValue(four)
2870     oEmitDouble(iAdd)        % addl $4, %esp
2871     oOperandPop
2872     oOperandPop
2873
2874     % Save Result
2875     @SaveResult
2876
2877     @RestoreTempRegsFromStack
2878
2879     % Force result into temp
2880     @ResultToTemp
2881     ;
```

coder.ssl

5.9 - Ord Operation

The ord operation needed to be updated to work with the string type. The new code moves a string address to a temp register. A trap is then called to return the corresponding integer in another temp register. The result is then saved and restored.

```
2884 OperandOrd:
2885
2886     % Push string descriptor onto stack
2887     @EmitStringDescriptor %lea s1, %T
2888                           %    %T
2889
2890     % Push temp on to stack
2891     @OperandPushTempWord   %    %X %T
2892
2893     % Push zero on to stack
2894     oOperandPushMode(mManifest)
2895     oOperandSetValue(zero)
2896     oOperandSetLength(word)  %  0 %X %T
2897
2898     % Set temp to zero
2899     oEmitDouble(iMov)            %   movl    $0, %X
2900     oOperandPop              %X=0 %T
2901
2902     % Mov first bye of word at [T] into temp
2903     oOperandSwap    %T X=0
2904     oOperandSetMode(mTempIndirect)
2905     oOperandSetLength(byte)
2906     oEmitDouble(iMov)   %movb (%T), %X
2907     @OperandPopAndFreeTemp;    %X
```

coder.ssl

5.10 - Equality Operation

A case for accepting the “tStringEqual” token was added to “OperandPushExpression”. The case then calls “OperandStringEqualPopPop”. This operation puts two string addresses into temporary registers. A trap call is then made which looks at the registers and determines if the contents at the addresses are equal. The strings are then popped from the OperandStack and the result is save then restored.

```
1552     | tStringEqual:
1553       @OperandStringEqualPop
```

```
3328 OperandStringEqualPop:
3329     % Generate code to compare string equality
3330
3331     @SaveTempRegsToStack
3332     @EmitStringDescriptor %lea s2, %T
3333     @OperandForceToStack  %pushl %T
3334     @OperandPopAndFreeTemp
3335     @EmitStringDescriptor %lea s1,  %T
3336     @OperandForceToStack %pushl, %T
3337     @OperandPopAndFreeTemp
3338
3339     % Make trap call
3340     oOperandPushMode(mTrap)
3341     oOperandSetValue(trStringEqual) % call pttrap106
3342     oOperandSetLength(word)
3343     oEmitSingle(iCall)
3344     oOperandPop
3345
3346     % Pop arguments
3347     oOperandPushMode(mStackReg)
3348     oOperandSetLength(word)
3349     oOperandPushMode(mManifest)
3350     oOperandSetLength(word)
3351     oOperandSetValue(eight)
3352     oEmitDouble(iAdd)
3353     oOperandPop
3354     oOperandPop
3355
3356     % Save result
3357     @SaveResult % movl eax, esi
3358
3359     @RestoreTempRegsFromStack
3360
3361     % Move result into temp
3362     @ResultToTemp;
```

coder.ssl

6 - Testing

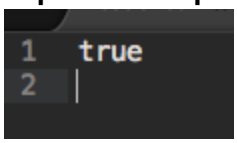
The coder phase was thoroughly tested using a variety of focused tests and provided example. An automation suite using rspec was also created to make it very simple to run tests quickly, compare outputs and find regressions.

All tests can be found in <project_root>/unit_tests/coder_tests

6.1 - If tests

If statements were tested for three cases: calling the if code, calling the elsif code and calling the else code. This was done with a simple if chain where the desired block printed true and the others printed false. The automation suite checks that the generated code prints true to verify the correct behaviour.

Test Files: if.pt, elseIf.pt, else.pt

Example Test:	**Expected Output:**
	

```
extern output

var a: integer
a = 2

if a == 2
    write("true")
else if a == 3
    write("false")
else
    write("false")
end

writeln

```

6.2 - Loop Tests

Loop statements were tested by initializing an array of strings and having a loop print out the elements and increment an index at each iteration. When the index reached the end of the array the loop exited. An expected output file was used to verify the output.

Test Files: loopTest.pt

Example Test:

```
extern output
var x: integer
var s : array [1 .. 12] of string

s[1] = "1"
s[2] = "2"
s[3] = "3"
s[4] = "4"
s[5] = "5"
s[6] = "6"
s[7] = "7"
s[8] = "8"
s[9] = "9"
s[10] = "10"
s[11] = "11"
s[12] = "12"

x=1
loop
    write(s[x])
    writeln
    x = x + 1
    break if x == 13
end

```

Expected Output:

```
1
2
3
4
5
6
7
8
9
10
11
12

```

6.3 - Switch Tests

Switch statements were tested for both the regular case and the default. If the correct block was executed, the word true would be printed - otherwise false.

Test Files: switch.pt, switchDefault.pt

Example Test:

```
extern output

var a: integer
a = 2

switch a
    case 1:
      write("false")
    case 2:
      write("true")
    default:
      write("false")
end

writeln
```

Expected Output:

```
true
```

6.4 - String Tests

Strings were tested thoroughly due to all the added features. Tests covered a variety of areas including: substrings, assignment, concatenation, array subscripting, equality operation, string parameters and empty strings.

Substrings

Substrings were tested by assigning a string variable and writing only a section of it. The test would pass if the correct substring was returned.

Example Test:

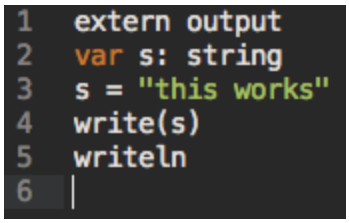
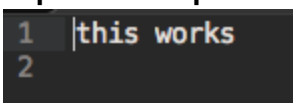
```
extern output

var s: string

s = "Hello World! Not!"

write(s::1..12)
writeln

```

Expected Output:

```
Hello World!

```

Assignment

Assignment was tested by creating simple string variables and verifying that writing the variable produced the correct value.

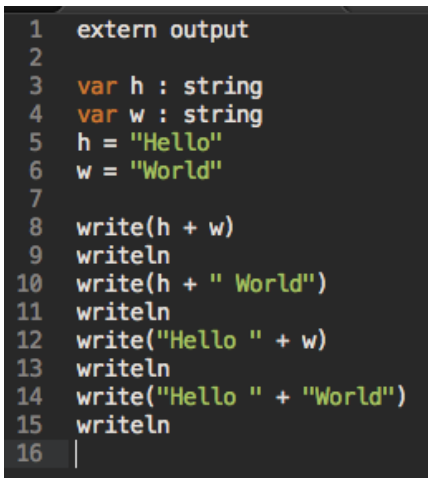
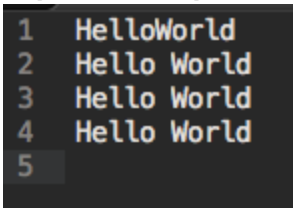
Example Test:	Expected Output:
	

```
extern output
var s: string
s = "this works"
write(s)
writeln
```

```
this works
```

Concatenation

Concatenation was tested by adding strings together in a variety of ways (eg. literal + variable, literal + literal, etc.). For each concatenation, the result was output to be verified.

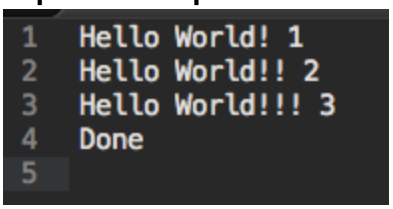
Example Test:	Expected Output:
	

```
extern output

var h : string
var w : string
h = "Hello"
w = "World"

write(h + w)
writeln
write(h + " World")
writeln
write("Hello " + w)
writeln
write("Hello " + "World")
writeln
```

```
HelloWorld
Hello World
Hello World
Hello World
```

Array Subscripting

Subscripting was done by creating an array of string variables and printing out the contents at each index.

Example Test:	Expected Output:
	

```
Hello World! 1
Hello World!! 2
Hello World!!! 3
Done
```

```
extern output

var sarray : array [1 .. 10] of string

sarray[1] = "Hello "
sarray[2] = "World"
sarray[3] = "! 1"
sarray[4] = "Hello "
sarray[5] = "World"
sarray[6] = "!! 2"
sarray[7] = "Hello "
sarray[8] = "World"
sarray[9] = "!!! 3"
sarray[10] = "Done"


write(sarray[1])
write(sarray[2])
write(sarray[3])
writeln
write(sarray[4])
write(sarray[5])
write(sarray[6])
writeln
write(sarray[7])
write(sarray[8])
write(sarray[9])
writeln
write(sarray[10])
writeln

```

Equality Operations

String equal and not equal were tested with an if statement that would output true if the comparison output the correct result.

Example Test:

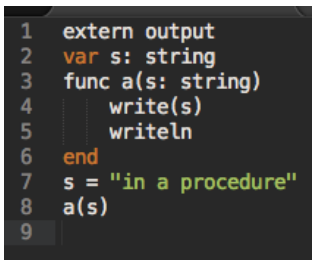
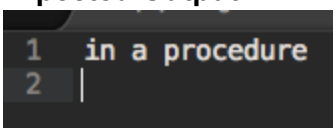
```
extern output

var s: string

s = "string"
if s == "string"
  write("true")
  writeln
else
  write("false")
  writeln
end

s = "not string"

if s != "string"
  write("true")
  writeln
else
  write("false")
  writeln
end

```

Expected Output:

```
true
true

```

String Parameters

String parameters were tested by creating a function that would accept a string and print it. If the parameter was passed correctly, an output would show.

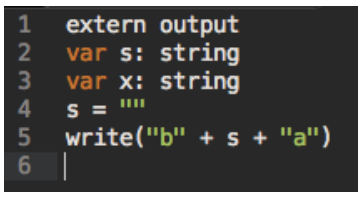
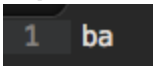
Example Test:	**Expected Output:**
	

```
extern output
var s: string
func a(s: string)
    write(s)
    writeln
end
s = "in a procedure"
a(s)
```

```
in a procedure
```

Empty String

The empty string was tested by adding it to other strings and ensuring the correct combined string was output.

Example Test:	**Expected Output:**
	

```
extern output
var s: string
var x: string
s = ""
write("b" + s + "a")
```

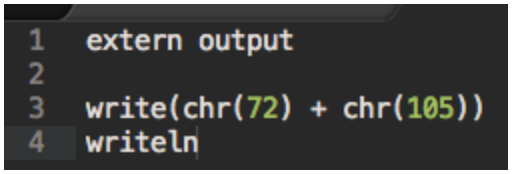
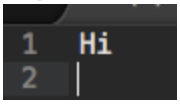
```
ba
```

6.5 - Chr and Ord

Chr was tested by outputting the sum of two chr operations and ensuring the correct string was produced.

Ord was tested by taking two Chrs and ensuring the correct codes were printed out.

Test Files: writeChar.pt, stringOrd.pt

Example Test:	**Expected Output:**
	

```
extern output

write(chr(72) + chr(105))
writeln
```

```
Hi
```

6.5 - Example Tests

The examples given by the course instructor cover the majority of the added functionality in complex and interconnected ways. To verify that all the components of our compiler were working together, all examples were compiled and executed. The versions compiled using our code can be found in: <project_root>/unit_tests/coder_tests/compiled_examples.

The example programs that only output were also added to the automation suite. These include: bubble.pt, lunch.pt, pascal.pt, primes.pt and stars.pt.

To ensure that examples with user input were not excluded, a special test case was also added for bust.pt.