Testing Documentation

How Testing Was Done

`ptsemtrace` and `semtrace`

semtrace and ptsemtrace are scripts we wrote to optimize getting the SSL trace output of our test files. They can be found under the scripts folder included in the submission. semtrace runs SSL trace using the Quby compiler library, whereas ptsemtrace runs SSL trace using the Pascal compiler library.

This allows us to compare the semantic token output of a Quby program to its equivalent Pascal program, allowing us to check for any errors.

The script usage is shortly described below:

```
semtrace <file> [<flag>]
    <file> : required : file address : file to ssltrace on
    <flag> : optional : string       : Flag to use to change trace behaviour

Default behaviour prints out emitted tokens.
```

Supported flags:

- `-ge`: Check the ssltrace output for errors using grep
- `-o`: Print emitted tokens and semantic operations (like trace in Tutorial 6)
- `-a`: Print entire trace (including branching and stuff)
- `-u`: Token output for default is automaticaally stripped, use this flag to keep unstripped
- Can also specify any other flag, which will be passed through to ssltrace e.g. `-i` to print input tokens

Iffy's Testing

All files and folders referred to in this section are under `ptsrc/test/phase-3/iffy`.

Handling String Literals

The following section consists of tests performed to verify that Quby treats Strings as first class data types and does not differentiate Strings and chars like Pascal. This will be done using the basic string equality, which is also tested.

To verify the proper handling of string literals for Quby, programs were defined in Pascal and Quby that handled a simple equality using string literals. It will be tested against the various types of literals.

Empty String

The file `stringops/string_lit_0_qb.pt` performs a string comparison using an empty string in Quby. `stringops/string_lit_0_pt.pt` does the same operation but in Pascal.

Performing ptsemtrace on the Pascal file, an error occurs as null/empty strings are not allowed in Pascal:

```
....
       oSymbolStkPush(syExpression)
       @StringLiteral
        oValuePushStringLength
        [ oValueChoose (zero)
        | zero:
        #eNullString
        semantic error, line 4: null literal string not allowed
        oTypeStkPush(tpChar)
        oTypeStkLinkToStandardType(stdChar)
...
```

Performing semtrace on the Quby file, a valid output token stream is gotten since null strings are allowed:

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
oEmitDataAddress
% value emitted 0
.tStoreInteger
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tLiteralString
oEmitString
.tLiteralString
oEmitString
.tStringEqual
.tAssignBoolean
.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

In the above output, string literals are handled with `.tLiteralString` and then the `oEmitString` semantic operation to emit the string value. This matches the handling of first-class data types in Quby, as seen by the output of `stringops/string_lit_int.pt` which performs a comparison with integer literals:

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
```

```
oEmitDataAddress
% value emitted 0
.tStoreInteger
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tLiteralInteger % literal code
oEmitValue % emits the value
% value emitted 0
.tLiteralInteger
oEmitValue
% value emitted 1
.tEQ
.tAssignBoolean
.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

1-Char String

The file `stringops/string_lit_1_qb.pt` performs a string comparison using a one-char string in Quby. `stringops/string_lit_1_pt.pt` does the same operation but in Pascal.

The output of ptsemtrace and semtrace on the Pascal file (left) and Quby file (right) respectively, are shown below:

.tFileDescriptor	.tFileDescriptor
.tLiteralInteger	.tLiteralInteger
oEmitValue	oEmitValue
% value emitted 2	% value emitted 2
.tFileBind	.tFileBind
.tLiteralAddress	.tLiteralAddress
oEmitDataAddress	oEmitDataAddress
% value emitted 0	% value emitted 0
.tStoreInteger	.tStoreInteger
.tAssignBegin	.tAssignBegin
.tLiteralAddress	.tLiteralAddress
oEmitValue	oEmitValue
% value emitted 4	% value emitted 4
.tLiteralChar	.tLiteralString
oEmitValue	oEmitString
% value emitted 97	% value emitted 97
.tLiteralChar	.tLiteralString
oEmitValue	oEmitString
% value emitted 97	% value emitted 97
.tEQ	.tStringEqual
.tAssignBoolean	.tAssignBoolean
.tTrapBegin	.tTrapBegin
.tTrap	.tTrap
oEmitTrapKind(trHalt)	oEmitTrapKind(trHalt)
% value emitted 0	% value emitted 0

In Pascal, one-char string literals are automatically converted to the Pascal char type. For Quby, they are seen as valid string and hence it continues to use the `tLiteralString` instead of the `tLiteralChar` used in

Pascal. Furthermore, it uses `tStringEqual` instead of `tEQ` as `semantic.ssl` uses that specific T-code for string equalities.

Also note that the 97, the ASCII code for `'a'` is emitted, which is the character in the source code.

Multi-char String

The file `stringops/string_lit_m_qb.pt` performs a string comparison using a multi-char string in Quby.

The semtrace output of this file is shown below:

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
oEmitDataAddress
% value emitted 0
.tStoreInteger
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tLiteralString
oEmitString
% value emitted 97
% value emitted 98
% value emitted 99
.tLiteralString
oEmitString
% value emitted 97
% value emitted 98
% value emitted 99
.tStringEqual
.tAssignBoolean
.tTrapBegin
.tTrap
oEmitTrapKind(trH
```

Each of the string literals begin with the `tLiteralString` token followed by `oEmitString` which emits the ASCII codes for the characters in the String (97 for `a`, 98 for `b` and 99 for `c`).

String Equality

String equality uses the special T-code `tStringEqual` instead of the standard `tEQ`, which is a change made from the Pascal semantic phase. To test this, the following files were written:

- `stringops/string_eq.pt` assigns a boolean variable the result of a simple string literal comparison (`"a" == "b"`)

- `stringops/string_eq_2.pt` assigns a boolean variable the result of a simple integer literal comparison

The output of semtrace for both files is shown below (`string_eq_2` on the left, and `string_eq` on the right):

.tFileDescriptor	.tFileDescriptor
.tLiteralInteger	.tLiteralInteger
oEmitValue	oEmitValue
% value emitted 2	% value emitted 2
.tFileBind	.tFileBind
.tLiteralAddress	.tLiteralAddress
oEmitDataAddress	oEmitDataAddress
% value emitted 0	% value emitted 0
.tStoreInteger	.tStoreInteger
.tAssignBegin	.tAssignBegin
.tLiteralAddress	.tLiteralAddress
oEmitValue	oEmitValue
% value emitted 4	% value emitted 4
.tLiteralInteger	.tLiteralString
oEmitValue	oEmitString
% value emitted 0	% value emitted 97
.tLiteralInteger	.tLiteralString
oEmitValue	oEmitString
% value emitted 1	% value emitted 98
.tEQ	.tStringEqual
.tAssignBoolean	.tAssignBoolean
.tTrapBegin	.tTrapBegin
.tTrap	.tTrap
oEmitTrapKind(trHalt)	oEmitTrapKind(trHalt)
% value emitted 0	% value emitted 0

As you can see in the above image, one of the differences in the output is the use of `tLiteralString` and `oEmitString` rather than the `tLiteralInteger` and `oEmitValue`. This makes sense since `string_eq` uses string literals rather than integer literals.

Furthermore, the `tStringEquals` token is used rather than the `tEQ`, showing that for String literals, the appropriate string token is used.

The differentiation is still made when using variables instead of literals, as indicated by the semtrace shown below (Left is `stringops/string_eq_vars_2.pt` which uses integer variables, right is `stringops/string_eq_vars.pt` which uses String variables):

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
oEmitDataAddress
% value emitted 0
.tStoreInteger
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tLiteralInteger
oEmitValue
% value emitted 0
.tAssignInteger
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 8
.tLiteralInteger
oEmitValue
% value emitted 1
.tAssignInteger
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 12
.tLiteralAddress
oEmitValue
% value emitted 4
.tFetchInteger
.tLiteralAddress
oEmitValue
% value emitted 8
.tFetchInteger
.tEQ
.tAssignBoolean
.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
oEmitDataAddress
% value emitted 0
.tStoreInteger
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tLiteralString
oEmitString
% value emitted 97
.tAssignString
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 1028
.tLiteralString
oEmitString
% value emitted 98
.tAssignString
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 2052
.tLiteralAddress
oEmitValue
% value emitted 4
.tFetchString
.tLiteralAddress
oEmitValue
% value emitted 1028
.tFetchString
.tStringEqual
.tAssignBoolean
.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

String Inequality

String inequality is implemented by inverting the output of the `tStringEquals` operation using the `tNot` T-code. This replaces the `tNEQ` for String comparisons.

This is indicated by the semtrace shown below:

- Left: `stringops/string_neq_2.pt`, performs integer literal inequality and assigns to variable
- Right: `stringops/string_neq.pt`, performs string literal inequality and assigns to variable

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
oEmitDataAddress
% value emitted 0
.tStoreInteger
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tLiteralInteger
oEmitValue
% value emitted 0
.tLiteralInteger
oEmitValue
% value emitted 1

.tNE
.tAssignBoolean
.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
oEmitDataAddress
% value emitted 0
.tStoreInteger
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tLiteralString
oEmitString
% value emitted 97
.tLiteralString
oEmitString
% value emitted 98
.tStringEqual
.tNot
.tAssignBoolean
.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

As seen above, the string inequality emits `tNot` after the `tStringEquals` to achieve the inequality operation on strings, rather than the `tNE` operation.

String Length

To test the semantic output of the String length operation, the file `stringops/string_len.pt` assigns the length of the string "abc" to an integer variable.

The semtrace output is shown below:

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
oEmitDataAddress
% value emitted 0
.tStoreInteger
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tLiteralString
oEmitString
% value emitted 97
% value emitted 98
% value emitted 99
.tLength
.tAssignInteger
```

```
.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

As seen above, the string literal is emitted first and is then followed by the `tLength` operation. Since the result type is an integer, the code `tAssignInteger` is used to assign it to an integer variable.

If the variable that is assigned to is not an integer type (as is the case in `stringops/string_len_invalid.pt`), a type clash will occur (as indicated with the `eTypeMismatch` error token):

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
oEmitDataAddress
% value emitted 0
.tStoreInteger
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tLiteralString
oEmitString
% value emitted 97
% value emitted 98
% value emitted 99
.tLength
#eTypeMismatch
.tAssignBoolean
.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

String Substring

`stringops/string_substr_valid_1.pt` was written to verify that the t-code output for the substring operation is correct, it performs a simple substring operation with only literals.

The semtrace output is shown below:

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
```

```
.tLiteralAddress
oEmitDataAddress
% value emitted 0
.tStoreInteger

.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tLiteralString
oEmitString
% value emitted 97
% value emitted 98
% value emitted 99
% value emitted 100
% value emitted 101
% value emitted 102
.tLiteralInteger
oEmitValue
% value emitted 2
.tLiteralInteger
oEmitValue
% value emitted 4
.tSubstring
.tAssignString

.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

As seen above, when the assignment statement begins the string literal is emitted along with the following integer literals. This is then followed by the `tSubstring` operation and then the `tAssignString` to end the assignment. All stacks are empty at the end of the code indicating correctness, and furthermore the output matches the format for other operations.

The substring operation also works with variables instead of literals. This was verified with the semtrace output of `stringops/string_substr_valid_2.pt` which uses a string variable and an integer variable for the substring operation:

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
oEmitDataAddress
% value emitted 0
.tStoreInteger
```

```
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tLiteralAddress
oEmitValue
% value emitted 1028
.tFetchString
.tLiteralInteger
oEmitValue
% value emitted 1
.tLiteralAddress
oEmitValue
% value emitted 2052
.tFetchInteger
.tSubstring
.tAssignString

.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

As seen above, the variables are identified with their address and fetch operations:

- The string variable is recognized with the `tLiteralAddress` and then the `tFetchString` operation
- The integer variable is recognized with the `tLiteralAddress` and then the `tFetchInteger` operation

Error detection was also verified as incorrect types cause a type clash error. This was tested with `stringops/string_substr_invalid_1.pt` which attempts to use a string literal for the starting index. The semtrace output shows the error:

```
...
      [ oTypeStkChooseKind (tpInteger)
      | tpInteger:
      oTypeStkPop
      [ oTypeStkChooseKind (tpString)
      | *:
      #eTypeMismatch
      semantic error, line 3: type clash
      oTypeStkPop
      oTypeStkPop
      ] or >
      >>
     ;HandleSubstringOperandTypeChecking
     oTypeStkPush(tpString)
     ] or >
     oSymbolStkPop
```

```
      oSymbolStkPop
...
```

The result type of the substring operation is also properly identified, as an attempt to assign it to a non-string variable (as done in `stringops/string_substr_invalid_2.pt`) results in a type clash error:

```
...
    | *:
    #eTypeMismatch
    semantic error, line 3: type clash
    ] or >
    >>
   ;CompareAndSwapTypes
   @EmitAssign
...
```

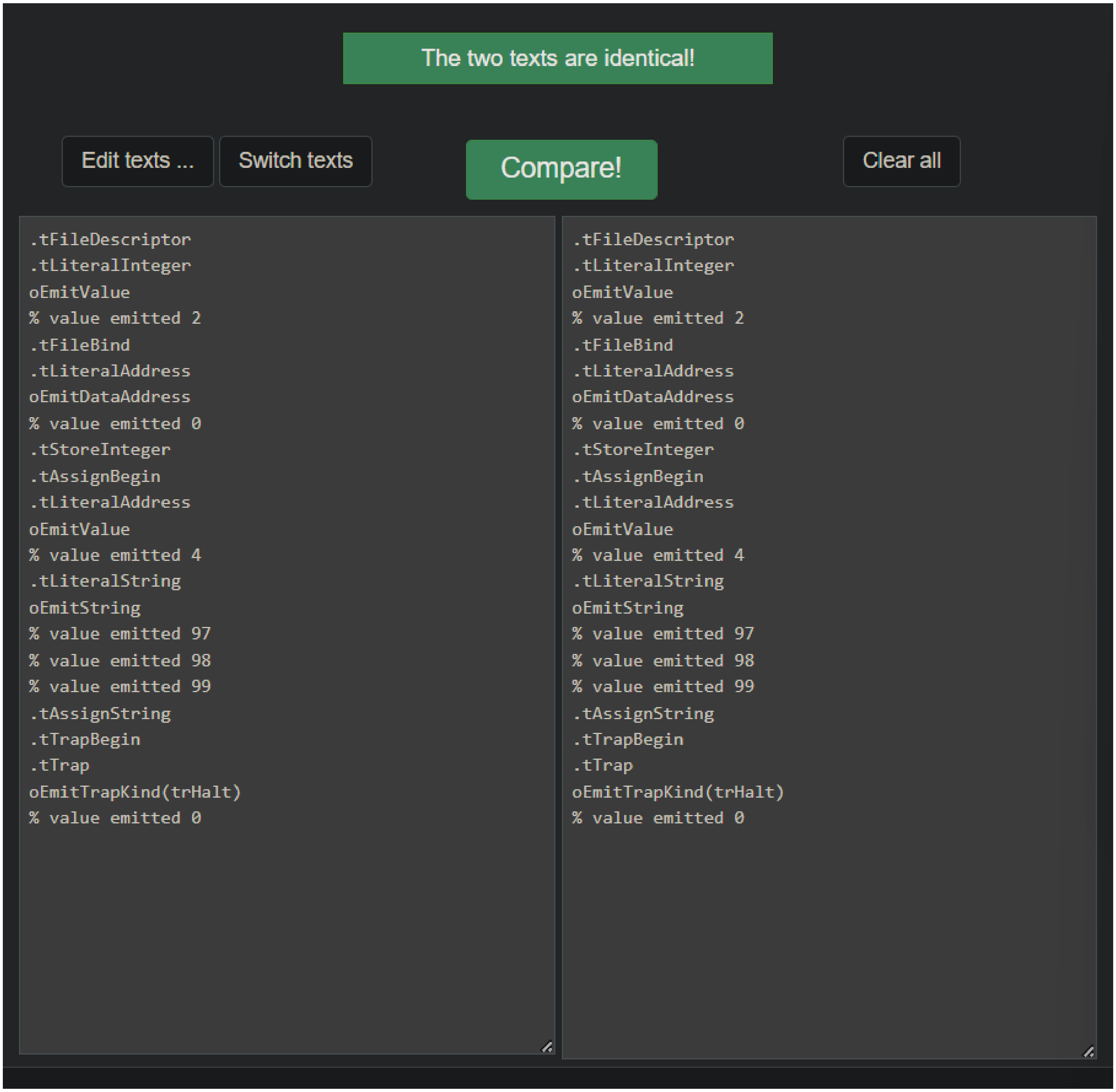
In both cases, there are no non-empty stacks which indicate proper error recovery.

String Constants

In Quby, string constants are handled in the same way as string variables.

The files `stringops/string_const.pt` and `stringops/string_var.pt` declare a string constant and string variable respectively. By comparing the `semtrace` output, the specified handling of string constants can be verified.

As seen by the text comparison below, the two output streams are identical:



Liam's Testing

All code used for testing can be found in /ptsrc/test/phase-3/liam/

`if elsif else` testing

In Quby, the `elsif` clause was added to if statements. By testing that the Quby and PT-pascal output the same tokens when compiling code with similar functionality allows us to ensure proper Quby functionality moving forward. The `if_test1.pt` is used with the Quby compiler and `if_test2.pt` runs with the PT-pascal compiler.

The Quby compiler had the following output:

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
oEmitDataAddress
% value emitted 0
.tStoreInteger
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tLiteralInteger
oEmitValue
% value emitted 1
.tAssignInteger
.tIfBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tFetchInteger
.tLiteralInteger
oEmitValue
% value emitted 1
.tEQ
.tIfThen
oEmitNullAddress
% value emitted -32767
.tWriteBegin
.tTrapBegin
.tLiteralAddress
oEmitValue
% value emitted 0
.tVarParm
.tParmEnd
.tLiteralAddress
oEmitValue
% value emitted 4
.tFetchInteger
.tParmEnd
.tLiteralInteger
oEmitValue
% value emitted 10
.tParmEnd
.tTrap
oEmitTrapKind(trWriteInteger)
% value emitted 8
```

```
.tWriteEnd
.tTrapBegin
.tLiteralAddress
oEmitValue
% value emitted 0
.tVarParm
.tParmEnd
.tTrap
oEmitTrapKind(trWriteln)
% value emitted 6
.tIfMerge
oEmitNullAddress
% value emitted -32767
.tIfBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tFetchInteger
.tLiteralInteger
oEmitValue
% value emitted 2
.tEQ
.tIfThen
oEmitNullAddress
% value emitted -32767
.tWriteBegin
.tTrapBegin
.tLiteralAddress
oEmitValue
% value emitted 0
.tVarParm
.tParmEnd
.tLiteralAddress
oEmitValue
% value emitted 4
.tFetchInteger
.tParmEnd
.tLiteralInteger
oEmitValue
% value emitted 10
.tParmEnd
.tTrap
oEmitTrapKind(trWriteInteger)
% value emitted 8
.tWriteEnd
.tTrapBegin
.tLiteralAddress
oEmitValue
```

```
% value emitted 0
.tVarParm
.tParmEnd
.tTrap
oEmitTrapKind(trWriteln)
% value emitted 6
.tIfEnd
.tIfEnd
.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

And the PT-pascal compiler outpetted the following:

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
oEmitDataAddress
% value emitted 0
.tStoreInteger
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tLiteralInteger
oEmitValue
% value emitted 1
.tAssignInteger
.tIfBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tFetchInteger
.tLiteralInteger
oEmitValue
% value emitted 1
.tEQ
.tIfThen
oEmitNullAddress
% value emitted -32767
.tWriteBegin
.tTrapBegin
.tLiteralAddress
oEmitValue
```

```
% value emitted 0
.tVarParm
.tParmEnd
.tLiteralAddress
oEmitValue
% value emitted 4
.tFetchInteger
.tParmEnd
.tLiteralInteger
oEmitValue
% value emitted 10
.tParmEnd
.tTrap
oEmitTrapKind(trWriteInteger)
% value emitted 8
.tWriteEnd
.tIfEnd
.tTrapBegin
.tLiteralAddress
oEmitValue
% value emitted 0
.tVarParm
.tParmEnd
.tTrap
oEmitTrapKind(trWriteln)
% value emitted 6
.tIfBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tFetchInteger
.tLiteralInteger
oEmitValue
% value emitted 2
.tEQ
.tIfThen
oEmitNullAddress
% value emitted -32767
.tWriteBegin
.tTrapBegin
.tLiteralAddress
oEmitValue
% value emitted 0
.tVarParm
.tParmEnd
.tLiteralAddress
oEmitValue
% value emitted 4
```

```
.tFetchInteger
.tParmEnd
.tLiteralInteger
oEmitValue
% value emitted 10
.tParmEnd
.tTrap
oEmitTrapKind(trWriteInteger)
% value emitted 8
.tWriteEnd
.tIfEnd
.tTrapBegin
.tLiteralAddress
oEmitValue
% value emitted 0
.tVarParm
.tParmEnd
.tTrap
oEmitTrapKind(trWriteln)
% value emitted 6
.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

As seen by the two outputs above, both returned similar T-tokens

`Module` Testing

The code for testing the module can be found in the file `module_test.pt` The file tests to see if the module declarations emit the correct values and store the module identifier on the stack so it can't be used once again.

Running the code outputted the following without errors:

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
oEmitDataAddress
% value emitted 0
.tStoreInteger
.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

`Public` Procedure Testing

Public procedures were a new form of procedure added to the new Quby language. Similarly to regular procedures, they emit the same T-tokens, however when placed on the symbol stack, the new `syPublicProcedure` symbol is used instead of the normally used `syProcedure` symbol. By delcaring it as a `syPublicProcedure` we can make it accessable outside of the scope of a module.

Running the `public_procedure_test.pt` file has a module with a public function declared within it. Running semtrace on it returned the following:

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
oEmitDataAddress
% value emitted 0
.tStoreInteger
.tSkipProc
oEmitNullAddress
% value emitted -32767
.tLiteralAddress
oEmitValue
% value emitted 4
.tStoreAddress
.tParmEnd
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tFetchAddress
.tLiteralAddress
oEmitValue
% value emitted 4
.tFetchAddress
.tFetchInteger
#eUndefinedIdentifier
#eExpnOperandReqd
.tAdd
.tAssignInteger
.tProcedureEnd
.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

Doing further inspection into each operation taking place and using the -a flag in with the semtrace script showed that the `syPublicProcedure` symbol is placed on the stack as opposed to the `syProcedure` symbol.

This will allow the procedure to be used outside of the module.

`Multi-Variable` Declaration Testing

Multi-Variable Decleration testing used the `multi_var_declaration_test1.pt` and `multi_var_decleration_test2.pt` files and was compiled using the ptsrc-ref compiler and the quby compiler being tested. The file `multi_var_declaration_test1.pt` was ran using the quby compiler and is expected to output the same T-tokens as the `multi_var_decleration_test2.pt` file should, when compiled using PT-pascal.

Output of test 1 file:

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
oEmitDataAddress
% value emitted 0
.tStoreInteger
.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

Running the same functionality code `multi_var_decleration_test2.pt` returned the same output, showing that the multivariable declaration functionality works.

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
oEmitDataAddress
% value emitted 0
.tStoreInteger
.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

Ethan's Testing

Procedure Scope Changes

As shown below, the changes made to the symbol table operations and block rule were effective. Scopes are pushed and pulled properly.

The source file for testing the block rule for procedures is `block_statement.pt`, located in `ptsrc/test/phase-3/ethan`.

```
oSymbolStkPush(syProcedure)
oSymbolTblPushScope
oCountPush(three)
 oSymbolStkPushLocalIdentifier
 oSymbolStkSetKind(syVariable)
 oTypeStkPush(tpFile)
 oTypeStkLinkToStandardType(stdText)
 oSymbolStkEnterTypeReference
 oValuePush(two)
  .tFileDescriptor
  oAllocateAlignOnWord
  oSymbolStkEnterDataAddress
  .tLiteralInteger
  oEmitValue
  % value emitted 2
  .tFileBind
  .tLiteralAddress
  oEmitDataAddress
  % value emitted 0
  .tStoreInteger
  oAllocateDescriptor
 oSymbolTblEnter
 oSymbolStkPop
 oTypeStkPop
 oValuePop
oCountPop
  oSymbolStkPushLocalIdentifier
  oSymbolStkSetKind(syConstant)
   oTypeStkPush(tpInteger)
   oTypeStkLinkToStandardType(stdInteger)
   oValuePushInteger
  oSymbolStkEnterTypeReference
  oTypeStkPop
  oSymbolStkEnterValue
  oValuePop
  oSymbolTblEnter
  oSymbolStkPop
  oSymbolStkPushLocalIdentifier
  oSymbolStkSetKind(syConstant)
   oTypeStkPush(tpInteger)
   oTypeStkLinkToStandardType(stdInteger)
   oValuePushInteger
  oSymbolStkEnterTypeReference
  oTypeStkPop
  oSymbolStkEnterValue
```

```
    oValuePop
    oSymbolTblEnter
    oSymbolStkPop
    oSymbolStkPushLocalIdentifier
    oSymbolStkSetKind(syProcedure)
    .tSkipProc
    oFixPushForwardBranch
    oEmitNullAddress
    % value emitted -32767
    oValuePushCodeAddress
    oSymbolStkEnterValue
    oValuePop
    oTypeStkPush(tpNull)
    oTypeStkSetRecursionFlag(yes)
    oTypeTblEnter
    oSymbolStkEnterTypeReference
    oSymbolTblEnter
    oSymbolTblPushScope
     oCountPush(zero)
     oCountIncrement
     oSymbolStkPushLocalIdentifier
     oSymbolStkSetKind(syVariable)
      oSymbolStkPushIdentifier
      oTypeStkPushSymbol
      oSymbolStkPop
      oSymbolStkEnterTypeReference
       oAllocateAlignOnWord
       oSymbolStkEnterDataAddress
       oAllocateVariable
      oSymbolTblEnter
     oCountIncrement
     oSymbolStkPushLocalIdentifier
     oSymbolStkSetKind(syVariable)
      oSymbolStkPushIdentifier
      oTypeStkPushSymbol
      oSymbolStkPop
      oSymbolStkEnterTypeReference
       oAllocateAlignOnWord
       oSymbolStkEnterDataAddress
       oAllocateVariable
      oSymbolTblEnter
     oCountIncrement
     oSymbolStkPushLocalIdentifier
     oSymbolStkSetKind(syVarParameter)
      oSymbolStkPushIdentifier
      oTypeStkPushSymbol
      oSymbolStkPop
      oSymbolStkEnterTypeReference
```

```
 oAllocateAlignOnWord
 oSymbolStkEnterDataAddress
 oAllocateVarParameter
oSymbolTblEnter
 oValuePushCount
 oCountPushValue
 oValuePop
.tLiteralAddress
oValuePushSymbol
oEmitValue
% value emitted 12
oValuePop
.tStoreAddress
oSymbolStkPop
oTypeStkPop
oCountDecrement
.tLiteralAddress
oValuePushSymbol
oEmitValue
% value emitted 8
oValuePop
 .tStoreInteger
oSymbolStkPop
oTypeStkPop
oCountDecrement
.tLiteralAddress
oValuePushSymbol
oEmitValue
% value emitted 4
oValuePop
 .tStoreInteger
oSymbolStkPop
oTypeStkPop
oCountDecrement
oCountPop
.tParmEnd
oTypeStkEnterParameterCount
oCountPop
oCountPush(one)
oSymbolStkPushLocalIdentifier
  oSymbolStkPushIdentifier
  oTypeStkPushSymbol
  oSymbolStkPop
oValuePushCount
oCountPushValue
 oCountDecrement
 oSymbolStkSetKind(syVariable)
  oAllocateAlignOnWord
```

```
      oSymbolStkEnterDataAddress
      oAllocateVariable
     oSymbolStkEnterTypeReference
     oSymbolTblEnter
    oCountPop
    oValuePop
    oTypeStkPop
    oCountDecrement
    oSymbolStkPop
    oCountPop
    oSymbolStkPushIdentifier
    .tAssignBegin
     .tLiteralAddress
     oValuePushSymbol
     oEmitValue
     % value emitted 16
     oValuePop
     oTypeStkPushSymbol
      oSymbolStkPushIdentifier
       .tLiteralAddress
       oValuePushSymbol
       oEmitValue
       % value emitted 4
       oValuePop
       oTypeStkPushSymbol
       .tFetchInteger
     oTypeStkSwap
     .tAssignInteger
    oTypeStkPop
    oSymbolStkPop
    oTypeStkPop
    oSymbolStkPop
   oTypeStkSetRecursionFlag(no)
   oTypeTblUpdate
   oTypeStkPop
   oSymbolTblUpdate
   oSymbolStkPop
   oSymbolTblPopScope
   oSymbolTblPreserveParameters
   .tProcedureEnd
   oFixPopForwardBranch
   oSymbolStkPushLocalIdentifier
   oSymbolStkSetKind(syPublicProcedure)
   .tSkipProc
   oFixPushForwardBranch
   oEmitNullAddress
   % value emitted -32767
   oValuePushCodeAddress
```

```
    oSymbolStkEnterValue
    oValuePop
    oTypeStkPush(tpNull)
    oTypeStkSetRecursionFlag(yes)
    oTypeTblEnter
    oSymbolStkEnterTypeReference
    oSymbolTblEnter
    oSymbolTblPushScope
     oCountPush(zero)
       oValuePushCount
       oCountPushValue
       oValuePop
      oCountPop
      .tParmEnd
      oTypeStkEnterParameterCount
      oCountPop
      oCountPush(one)
      oSymbolStkPushLocalIdentifier
        oSymbolStkPushIdentifier
        oTypeStkPushSymbol
        oSymbolStkPop
      oValuePushCount
      oCountPushValue
       oCountDecrement
       oSymbolStkSetKind(syVariable)
        oAllocateAlignOnWord
        oSymbolStkEnterDataAddress
        oAllocateVariable
       oSymbolStkEnterTypeReference
       oSymbolTblEnter
      oCountPop
      oValuePop
      oTypeStkPop
      oCountDecrement
      oSymbolStkPop
      oCountPop
    oTypeStkSetRecursionFlag(no)
    oTypeTblUpdate
    oTypeStkPop
    oSymbolTblUpdate
    oSymbolStkPop
    oSymbolTblPopScope
    oSymbolTblPreserveParameters
    .tProcedureEnd
    oFixPopForwardBranch
  oSymbolTblPopScope
  oSymbolStkPop
  .tTrapBegin
```

```
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

String Index Operator

This section corresponds to the String Index operator (?). Testing is completed for string literals and variables, and for semantically correct and incorrect source code.

Literals

Semantically Correct Test

Below is the output for a semantically correct String Index operation on two string literals, in source file `sti_lit_valid.pt`.

As can be seen in the tokens emitted, the assignment is properly recognized as semantically correct.

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
oEmitDataAddress
% value emitted 0
.tStoreInteger
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tLiteralString
oEmitString
% value emitted 72
% value emitted 101
% value emitted 108
% value emitted 108
% value emitted 111
% value emitted 32
% value emitted 116
% value emitted 104
% value emitted 101
% value emitted 114
% value emitted 101
.tLiteralString
oEmitString
% value emitted 116
% value emitted 104
% value emitted 101
```

```
.tIndex
.tAssignInteger
.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

Semantically Incorrect Test

Below is the error output for a semantically incorrect String Index operation on a string literal and an integer literal, in source file `sti_lit_invalid.pt`.

The compiler correctly identifies that the String Index operation cannot be performed between an integer and a string.

```
semantic error, line 5: operand and operator types clash
```

Variables

Semantically Correct Test

Below is the output for a semantically correct String Index operation on two string variables assigned to an integer variable, in source file `sti_var_valid.pt`.

As can be seen in the tokens emitted, the assignment operation is properly recognized as semantically correct.

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
oEmitDataAddress
% value emitted 0
.tStoreInteger
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tLiteralString
oEmitString
% value emitted 72
% value emitted 101
% value emitted 108
% value emitted 108
% value emitted 111
% value emitted 32
% value emitted 116
```

```
% value emitted 104
% value emitted 101
% value emitted 114
% value emitted 101
.tAssignString
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 1028
.tLiteralString
oEmitString
% value emitted 116
% value emitted 104
% value emitted 101
.tAssignString
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 2052
.tLiteralAddress
oEmitValue
% value emitted 4
.tFetchString
.tLiteralAddress
oEmitValue
% value emitted 1028
.tFetchString
.tIndex
.tAssignInteger
.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

Semantically Incorrect Test

Below is the error output for a semantically incorrect String Index operation on a string variable and an integer constant to an integer variable, in source file `sti_var_invalid.pt`.
The compiler correctly identifies that the String Index operation cannot be performed between an integer and a string.

```
semantic error, line 10: operand and operator types clash
```

String Concatenation

This section corresponds to String Concatenation. Testing is completed for string literals and variables, and for semantically correct and incorrect source code.

Literals

Semantically Correct Test

Below is the output for a semantically correct string concatenation on two string literals, in source file `stcat_lit_valid.pt`.

As can be seen in the tokens emitted, the concatenation and assignment is properly recognized as semantically correct.

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
oEmitDataAddress
% value emitted 0
.tStoreInteger
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tLiteralString
oEmitString
% value emitted 72
% value emitted 101
% value emitted 108
% value emitted 108
% value emitted 111
% value emitted 32
.tLiteralString
oEmitString
% value emitted 87
% value emitted 111
% value emitted 114
% value emitted 108
% value emitted 100
.tConcatenate
.tAssignString
.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

Semantically Incorrect Test

Below is the error output for a semantically incorrect string concatenation on a string literal and an integer literal, in source file `stcat_lit_invalid.pt`.

The compiler correctly identifies that an integer and a string cannot be added.

```
semantic error, line 5: type clash
```

Variables

Semantically Correct Test

Below is the output for a semantically correct string concatenation on two string variables assigned to an integer variable, in source file `stcat_var_valid.pt`.

As can be seen in the tokens emitted, the concatenation and assignment is properly recognized as semantically correct.

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
oEmitDataAddress
% value emitted 0
.tStoreInteger
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tLiteralString
oEmitString
% value emitted 72
% value emitted 101
% value emitted 108
% value emitted 108
% value emitted 111
% value emitted 32
.tAssignString
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 1028
.tLiteralString
oEmitString
% value emitted 87
% value emitted 111
% value emitted 114
% value emitted 108
% value emitted 100
.tAssignString
.tAssignBegin
.tLiteralAddress
oEmitValue
```

```
% value emitted 2052
.tLiteralAddress
oEmitValue
% value emitted 4
.tFetchString
.tLiteralAddress
oEmitValue
% value emitted 1028
.tFetchString
.tConcatenate
.tAssignString
.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

Semantically Incorrect Test

Below is the error output for a semantically incorrect string concatenation on a string variable and an integer constant to an integer variable, in source file `stcat_var_invalid.pt`.

The compiler correctly identifies that an integer and a string cannot be added.

```
semantic error, line 10: type clash
```

Type Definitions

Semantically Correct Test

Below is the output for a semantically correct type definition, in source file `type_valid.pt`.
The type is correctly identified and added to the symbol table.

```
 oSymbolStkPush(syProcedure)
 oSymbolTblPushScope
 oCountPush(three)
  oSymbolStkPushLocalIdentifier
  oSymbolStkSetKind(syVariable)
  oTypeStkPush(tpFile)
  oTypeStkLinkToStandardType(stdText)
  oSymbolStkEnterTypeReference
  oValuePush(two)
   .tFileDescriptor
   oAllocateAlignOnWord
   oSymbolStkEnterDataAddress
   .tLiteralInteger
   oEmitValue
   % value emitted 2
   .tFileBind
   .tLiteralAddress
```

```
    oEmitDataAddress
    % value emitted 0
    .tStoreInteger
    oAllocateDescriptor
   oSymbolTblEnter
   oSymbolStkPop
   oTypeStkPop
   oValuePop
  oCountPop
    oSymbolStkPushLocalIdentifier
    oSymbolStkSetKind(syType)
      oSymbolStkPushIdentifier
      oTypeStkPushSymbol
      oSymbolStkPop
    oSymbolStkEnterTypeReference
    oTypeStkPop
    oSymbolTblEnter
    oSymbolStkPop
  oSymbolTblPopScope
  oSymbolStkPop
  .tTrapBegin
  .tTrap
  oEmitTrapKind(trHalt)
  % value emitted 0
```

Semantically Incorrect Test

Below is the error output for a semantically incorrect type definition, in source file `type_invalid.pt`.
The compiler recognizes that only one type can be declared per definition.

```
scan/parse error, line 3: syntax error at: ,
```

Noah's Testing

Procedure

All of the files being tested in this document are located under 'test/phase-3/noah'.

`case` statement

The changes made to the case statement in this semantic phase only concern the generation of the else clause after the case. Otherwise, the same behaviour as PT Pascal is mimicked with the new syntax in the previous phase of the compiler.

Following is the output from the file 'case_1.pt' that shows a valid example of a case statement and its output using the Quby compiler.

```
.tFileDescriptor
.tLiteralInteger
```

```
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
oEmitDataAddress
% value emitted 0
.tStoreInteger
.tCaseBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tFetchInteger
.tCaseSelect
oEmitNullAddress
% value emitted -32767
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 8
.tLiteralInteger
oEmitValue
% value emitted 7
.tAssignInteger
.tCaseMerge
oEmitNullAddress
% value emitted -32767
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 8
.tLiteralInteger
oEmitValue
% value emitted 8
.tAssignInteger
.tCaseMerge
oEmitNullAddress
% value emitted -32767
.tCaseEnd
.tCaseElse
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 8
.tLiteralInteger
oEmitValue
% value emitted 9
.tAssignInteger
.tCaseMerge
```

```
oEmitNullAddress
% value emitted -32767
oEmitCaseBranchTable
% value emitted 6
% value emitted 7
% value emitted 19
% value emitted 31
.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

This as we can see, is very similar to the output of the similar PTPascal test file, 'pt_case_1.pt', with the notable exception of the `.tCaseElse` and all corresponding functionality after the `.tCaseEnd` token.

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
oEmitDataAddress
% value emitted 0
.tStoreInteger
.tCaseBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tFetchInteger
.tCaseSelect
oEmitNullAddress
% value emitted -32767
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 8
.tLiteralInteger
oEmitValue
% value emitted 8
.tAssignInteger
.tCaseMerge
oEmitNullAddress
% value emitted -32767
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 8
.tLiteralInteger
```

```
oEmitValue
% value emitted 9
.tAssignInteger
.tCaseMerge
oEmitNullAddress
% value emitted -32767
.tCaseEnd
oEmitCaseBranchTable
% value emitted 6
% value emitted 7
% value emitted 17
% value emitted 27
.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

`do` statement

Unlike a lot of the other statements, the new `do` statement in Quby doesn't have a direct correlary in PTPascal. The most similar thing that we can compare it to is a `while` loop.

The below output token stream is from the test file 'do_1.pt'.

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
oEmitDataAddress
% value emitted 0
.tStoreInteger
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tLiteralInteger
oEmitValue
% value emitted 1
.tAssignInteger
.tDoBegin
.tDoBreakIf
.tLiteralAddress
oEmitValue
% value emitted 4
.tFetchInteger
.tLiteralInteger
oEmitValue
```

```
% value emitted 42
.tLT
.tDoTest
oEmitNullAddress
% value emitted -32767
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tLiteralAddress
oEmitValue
% value emitted 4
.tFetchInteger
.tLiteralInteger
oEmitValue
% value emitted 1
.tAdd
.tAssignInteger
.tDoEnd
% value emitted 22
.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

This output is exactly as expected. The do statement starts with the emission of a `.tDoBegin` and the `break if` statements are bounded by `.tDoBreakIf` and `.tDoTest`. To finish it off, the loop is ended with a `.tDoEnd`. A similar PTPascal example with a while loop is present in 'pt_do_1.pt'. The output when compiled with the PTPascal compiler is shown below.

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
oEmitDataAddress
% value emitted 0
.tStoreInteger
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tLiteralInteger
oEmitValue
% value emitted 1
.tAssignInteger
.tWhileBegin
```

```
.tLiteralAddress
oEmitValue
% value emitted 4
.tFetchInteger
.tLiteralInteger
oEmitValue
% value emitted 42
.tLT
.tWhileTest
oEmitNullAddress
% value emitted -32767
.tAssignBegin
.tLiteralAddress
oEmitValue
% value emitted 4
.tLiteralAddress
oEmitValue
% value emitted 4
.tFetchInteger
.tLiteralInteger
oEmitValue
% value emitted 1
.tAdd
.tAssignInteger
.tWhileEnd
% value emitted 20
.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

This is very similar to the `do` example with some exceptions. In this example, a `.tWhileBegin` and `.tWhileEnd` bound the loop. As well the condition is only followed by a T-Code token `.tWhileTest` instead of being bounded on either side since its position is predictable in regular PTPascal while loops. This correlary shows that the do loop is indeed outputting the correct tokens for the semantic phase.

Multiple constant declarations

In Quby, defining multiple constants using a single constant keyword (now `val` instead of `const`) is disallowed. The modification to this rule was successful as we can see that the regular single constant per definition works as shown in 'constants_1.pt'. The following is the output:

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
oEmitDataAddress
```

```
% value emitted 0
.tStoreInteger
.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```

Meanwhile, as expected, declaring multiple constants in one line does not work as can be shown in 'bad_constants_1.pt'. Running this file with the Quby compiler fails with `#eUndefinedIdentifier` since the subsequent identifier in the same line is no longer valid.

```
.tFileDescriptor
.tLiteralInteger
oEmitValue
% value emitted 2
.tFileBind
.tLiteralAddress
oEmitDataAddress
% value emitted 0
.tStoreInteger
.tAssignBegin
#eUndefinedIdentifier
.tLiteralInteger
oEmitValue
% value emitted 10
.tAssignInteger
.tTrapBegin
.tTrap
oEmitTrapKind(trHalt)
% value emitted 0
```