Testing Documentation

All files referred to are in ptsrc/test/phase-2.

The testing document also uses the term parsetrace, this is a short form referring to using the SSL trace command on our built compiler up to the parser phase.

For example, running parsetrace on test.pt is equivalent to:

```
ssltrace "ptc -o2 -t2 -L CMPE458/ptsrc/lib/pt test.pt" CMPE458/ptsrc/lib/pt/parser.def -e
```

This was implemented with a shell script.

The order is as follows:

- Iffy
- Ethan
- Noah
- Liam

Iffy's Testing

String Operations

All test files mentioned in this file are located in the folder ptsrc/test/phase-2/iffy/stringops.

Length Operation

We use `string_length.pt` to test the parsing of the string length operation as a standalone assignment statement. We simply assign the length of the variable `str` to the variable `a`.

The expected output would be:

```
.sProgram        // For the pUsing token
.sIdentifier     // Identifier of the using statement
.sParmEnd        // Part of the program definition
.sBegin          // Block begin statement

.sAssignmentStmt // assignment token
.sIdentifier     // identifier for a
.sIdentifier     // identifier for str
.sLength    // length operator token after to follow postfix notation
.sExpnEnd   // end expression token
```

```
.sEnd            // Block end statement
```

This is verified by the actual output of the test file from parsetrace:

```
 .sProgram
 % .sNewLine
 .sIdentifier
 .sParmEnd
  .sBegin
   .sAssignmentStmt
   .sIdentifier
             .sIdentifier
        .sLength
   .sExpnEnd
  .sEnd
```

Similar output would be gotten for the use of a string literal instead of a variable (see `string_length_literal.pt`), the only difference being the `sIdentifier` token for the `str` variable will be replaced with an `sStringLiteral` token.

```
 .sProgram
 % .sNewLine
 % .sNewLine
 .sIdentifier
 .sParmEnd
  .sBegin
   .sAssignmentStmt
   .sIdentifier
             .sStringLiteral // replaced sIdentifier
        .sLength
   .sExpnEnd
  .sEnd
```

This would apply for other string operation and testing.

The length operation was also tested to be able to parse expressions. This was done with `string_length_exprs.pt` which replaces the static string with a simple concatenation.

Following post fix notation, the expected output token stream for the expression would be:

```
sStringLiteral // "Hel" string
sStringLiteral // "lo" string
sAdd // Add the strings together
sLength // the length operation
```

This is the same as the actual parsetrace output:

```
.sProgram
% .sNewLine
.sIdentifier
.sParmEnd
 .sBegin
  .sAssignmentStmt
  .sIdentifier
            .sStringLiteral
            .sStringLiteral
         .sAdd
       .sLength
  .sExpnEnd
 .sEnd
```

Note that the brackets are required because the Factor rule is called instead of the Expression rule in order to make sure precedence rules are obeyed. If the expression rule is called, lower binding operators can be binded before the current operator.

Substring Operation

The test file `string_substring.pt` tests the parsing of the substring operator on a string literal given integer constant ranges. The expected output would be

```
.sProgram        // For the pUsing token
.sIdentifier     // Identifier of the using statement
.sParmEnd        // Part of the program definition
.sBegin          // Block begin statement

.sAssignmentStmt // assignment token
.sIdentifier     // identifier for a
.sStringLiteral // String variable
.sInteger // Integer start subscript 1
.sInteger // Integer end subscrtipt 3
.sSubstring // emitted following post fix notation
.sExpnEnd   // end expression token

.sEnd            // Block end statement
```

This is verified by the actual output from parsetrace:

```
.sProgram
% .sNewLine
% .sNewLine
```

```
 .sIdentifier
 .sParmEnd
  .sBegin
   .sAssignmentStmt
   .sIdentifier
        .sStringLiteral
            .sInteger
            .sInteger
       .sSubstring
   .sExpnEnd
  .sEnd
```

The file `string_substring_exprs.pt` is used to test that expressions can be used for the string subscript values. It performs simple integer calculations to determine the range. The expected output would be the post fix notation for each subscript expression.

So for the range specification: `(1+2) .. (3*4)`, we expect the output to be:

```
.sStringLiteral // The string we are subscripting
.sInteger // The integer 1
.sInteger // The integer 2
.sAdd // Do 1+2
.sInteger // The integer 3
.sInteger // The integer 4
.sMultiply // Do 3*4
.sSubstring
```

This is verified by the parsetrace on `string_substring_exprs.pt`:

```
.sProgram
 % .sNewLine
 % .sNewLine
 .sIdentifier
 .sParmEnd
  .sBegin
   .sAssignmentStmt
   .sIdentifier
        .sStringLiteral
            .sInteger
            .sInteger
         .sAdd
            .sInteger
            .sInteger
          .sMultiply
       .sSubstring
```

```
    .sExpnEnd
   .sEnd
```

Note that the brackets are required because the Factor rule is called instead of the Expression rule in order to make sure precedence rules are obeyed. If the expression rule is called, lower binding operators can be binded before the current operator.

Index Operation

The file `string_index.pt` tests the how the string index operation when the `?` operator is used. Similar to `string_substring.pt` it performs the test with two static string literals. The expected output would be:

```
.sProgram        // For the pUsing token
.sIdentifier     // Identifier of the using statement
.sParmEnd        // Part of the program definition
.sBegin          // Block begin statement

.sAssignmentStmt // assignment token
.sIdentifier     // identifier for i
.sStringLiteral // "Hello World" string
.sStringLiteral // "Hello" string we are subscripting
.sIndex // The index operator emitted afterward following postfix notation
.sExpnEnd   // end expression token

.sEnd            // Block end statement
```

This is verified by the parsetrace output on the test file:

```
  .sProgram
  % .sNewLine
  % .sNewLine
  .sIdentifier
  .sParmEnd
   .sBegin
    .sAssignmentStmt
    .sIdentifier
         .sStringLiteral
         .sStringLiteral
       .sIndex
    .sExpnEnd
   .sEnd
```

The index operator also can parse expressions, and this can be verified by the parsetrace output of `string_index_exprs.pt`, which simply replaces the static index string with a simple concatenation.

```
 .sProgram
 % .sNewLine
 % .sNewLine
 .sIdentifier
 .sParmEnd
  .sBegin
   .sAssignmentStmt
   .sIdentifier
        .sStringLiteral
           .sStringLiteral // "Hel" string
           .sStringLiteral // "lo" string
        .sAdd // Add to add the tokens together, follows post fix notation
      .sIndex // Index following post fix notation
   .sExpnEnd
  .sEnd
```

Testing for Precedence

Testing Precedence(`#`) > Precedence(`?`)

`string_precedence_1.pt` verifies that `#` has higher precedence than the `?`. It features a simple assignment `a = # "Hello World" ? "Hello"` which does not use any brackets.

Following the precedence rules, the `#` should be binded first and then the `?`. This would result in the following token stream:

```
.sStringLiteral // String "Hello World"
.sLength // Length operator bound to "Hello World"
.sStringLiteral // String "Hello"
.sIndex // Index operator performing on (# "Hello World") and "Hello"
```

This is verified by the parsetrace output:

```
 .sProgram
 % .sNewLine
 % .sNewLine
 .sIdentifier
 .sParmEnd
  .sBegin
   .sAssignmentStmt
   .sIdentifier
         .sStringLiteral
        .sLength
        .sStringLiteral
      .sIndex
```

```
    .sExpnEnd
   .sEnd
```

Note that this program and assignment is technically illegal, however this would be identified by the semantic analysis stage. The parser is context free and therefore does not know the hash operation returns an integer.

We can force the `?` to have higher precedence by surrounding the expression in brackets, since the Factor rule parses contents in brackets as expressions. This is what is done in `string_precedence_1_alt.pt`, with the output shown below:

```
  .sProgram
  % .sNewLine
  .sIdentifier
  .sParmEnd
   .sBegin
    .sAssignmentStmt
    .sIdentifier
               .sStringLiteral
               .sStringLiteral
             .sIndex // index operation first
         .sLength // Then length operation next
    .sExpnEnd
   .sEnd
```

Testing Precedence(`$`) > Precedence(`?`)

`string_precedence_2.pt` tests that the precedence of `$` is higher than that of the `?`, with the use of the statement `i = "Hello World" ? "Hello" $ 1 .. 2`.

Following precedence rules, the `$` should bind to the "Hello" first, and then the `?`. This would generate the token stream:

```
.sStringLiteral // "Hello World"
.sStringLiteral // "Hello"
.sInteger // 1
.sInteger // 2
.sSubstring // appears first to do "Hello" $ 1 .. 2
.sIndex // next to do ? on result i.e. "Hello World" ? ("Hello" $ 1 .. 2)
```

This is verified by the parsetrace output below:

```
  .sProgram
  % .sNewLine
  .sIdentifier
```

```
.sParmEnd
 .sBegin
  .sAssignmentStmt
  .sIdentifier
       .sStringLiteral
          .sStringLiteral
          .sInteger
          .sInteger
         .sSubstring
     .sIndex
  .sExpnEnd
 .sEnd
```

We can force the `?` to have higher precedence by surrounding the expression in brackets. This is what is done in `string_precedence_2_alt.pt`, with the output shown below:

```
.sProgram
% .sNewLine
.sIdentifier
.sParmEnd
 .sBegin
  .sAssignmentStmt
  .sIdentifier
            .sStringLiteral
               .sStringLiteral
          .sIndex // Index operation is performed first
       .sInteger
       .sInteger
      .sSubstring // Then the substring operation
  .sExpnEnd
 .sEnd
```

Testing Precedence(`#`) > Precedence(`$`)

`string_precedence_3.pt` tests that the precedence of `#` is higher than that of the `$`, with the use of the statement `i = # "Hello World" $ 1 .. 4`.

Following precedence rules, the `#` should bind to the "Hello World" first, and then the `$`. This would generate the token stream:

```
.sStringLiteral // "Hello World"
.sLength // length operation on "Hello World"
.sInteger // 1
.sInteger // 2
.sSubstring // substring on result i.e. ( # "Hello World" ) $ 1 .. 4
```

This is verified by the parsetrace output below:

```
 .sProgram
 % .sNewLine
 % .sNewLine
 .sIdentifier
 .sParmEnd
  .sBegin
   .sAssignmentStmt
   .sIdentifier
         .sStringLiteral
        .sLength
        .sInteger
        .sInteger
       .sSubstring
   .sExpnEnd
  .sEnd
```

We can force the `$` to have higher precedence by surrounding the expression in brackets. This is what is done in `string_precedence_3_alt.pt`, with the output shown below:

```
 .sProgram
 % .sNewLine
 % .sNewLine
 .sIdentifier
 .sParmEnd
  .sBegin
   .sAssignmentStmt
   .sIdentifier
              .sStringLiteral
              .sInteger
              .sInteger
             .sSubstring // substring binded first
        .sLength // then length operation
   .sExpnEnd
  .sEnd
```

Minor Syntactic Details

All test files mentioned in this file are located in the folder ptsrc/test/phase-2/iffy/minordetails.

Assign

These tests are just used to verify that the new token for the assignment operator, as well as the changed tokens for not, not equals and comparisons are correctly parsed by the system.

The file `parsing_assign.pt` is used to test that the new assignment operator `=` generates the `sAssignmentStmt` token, as this will show that the parser recognizes the new assignment token. This is verified by the parsetrace output below:

```
 .sProgram
 % .sNewLine
 .sIdentifier
 .sParmEnd
  .sBegin
   .sAssignmentStmt
   .sIdentifier
        .sIdentifier
   .sExpnEnd
  .sEnd
```

If the old colon assign is used (`:=`) as seen in `parsing_assign_invalid.pt`, a parsing error occurs because of the colon:

```
...
  ;AssignmentOrCallStmt
  ] or >
  }
  [ (pColon)
  | *:
  ] or >
  .sEnd
  >>
 ;Block
 >>
;Program
scan/parse error, line 2: syntax error at: :
```

Not and Not Equals

The test file `parsing_not.pt` tests that the new not operator `!` is parsed by the parser. This is verified by its parsetrace output:

```
 .sProgram
 % .sNewLine
 .sIdentifier
 .sParmEnd
  .sBegin
   .sAssignmentStmt
   .sIdentifier
         .sIdentifier
```

```
        .sNot
    .sExpnEnd
   .sEnd
```

As seen above, the `sNot` semantic token is emitted when the new not operator is used. If the old not keyword is used (as seen in `parsing_not_invalid.pt`), we get semantically illegal parsing:

```
  .sProgram
  % .sNewLine
  .sIdentifier
  .sParmEnd
   .sBegin
    .sAssignmentStmt
    .sIdentifier
         .sIdentifier
    .sExpnEnd
     .sCallStmt
     .sIdentifier
     .sParmEnd
   .sEnd
```

The `not` is parsed as an identifier and is marked as the full assignment statement, while the actual variable to invert is recognized as a call statement. This does not fail in parsing, but will fail in the semantic analysis stage.

Comparison

As we have already seen that the old comparison operator `=` is parsed as an assignment statement, we simply need just one test file `parsing_compare.pt` which tests that `==` is now the comparison operator.

Its parse trace output is shown below:

```
  .sProgram
  % .sNewLine
  .sIdentifier
  .sParmEnd
   .sBegin
    .sAssignmentStmt
    .sIdentifier
              .sIdentifier
              .sIdentifier
          .sEq
    .sExpnEnd
   .sEnd
```

As seen, the `b == c` is correctly parsed into the two `sIdentifier` tokens followed by the `sEq` statement.

Declarations

All test files mentioned in this file are located in the folder ptsrc/test/phase-2/iffy/declarations.

General Invalid Assignments

Missing Identifier

If the declaration is missing an identifier (for constant, type and variable declarations), the parser will throw an error as all rules are expecting the `pIdentifier` token after the keyword. Instead of that they receive the token for the declaration assignment operator (either colon or `=`)

This is verified by the parser trace output on the file `general_invalid_1.pt`, which omits the identifier for a variable declaration.

```
...
.sVar
  @VariableDeclarations
   ?pIdentifier (pColon)
   scan/parse error, line 3: syntax error at: :
   .sIdentifier
   [ (pColon)
   | *:
...
```

The same error will occur for other declaration types as based on `general_invalid_2.pt` and `general_invalid_3.pt`

`general_invalid_2.pt` output:

```
...
  @ConstantDefinitions
   ?pIdentifier (pAssignEquals)
   scan/parse error, line 3: syntax error at: =
   .sIdentifier
   ?pAssignEquals (pAssignEquals)
   @ConstantValue
...
```

`general_invalid_3.pt` output:

```
...
  @TypeDefinitions
```

```
    ?pIdentifier (pColon)
    scan/parse error, line 3: syntax error at: :
    .sIdentifier
    ?pColon (pColon)
    @TypeBody
     [ (pColon)
     | *:
     @SimpleType
...
```

Keyword as Identifier Name

If a keyword is used as the identifier name for any declaration, the parser will thrown an error. This is because keywords have their own tokens and are not recognized as `pIdentifier` tokens, which are what the declaration rules are expecting.

This is verified by the parser trace output on the file `general_invalid_4.pt`, which uses the unless keyword for a variable declaration.

```
...
  @VariableDeclarations
   ?pIdentifier (pUnless)
   scan/parse error, line 3: syntax error at: unless
   .sIdentifier
   [ (pUnless)
   | *:
   ] or >
...
```

he same error will occur for other declaration types as based on `general_invalid_5.pt` and `general_invalid_6.pt`

`general_invalid_5.pt` output:

```
...
  @ConstantDefinitions
   ?pIdentifier (pUnless)
   scan/parse error, line 3: syntax error at: unless
   .sIdentifier
   ?pAssignEquals (pUnless)
   @ConstantValue
    [ (pUnless)
...
```

`general_invalid_6.pt` output:

```
...
 @TypeDefinitions
   ?pIdentifier (pUnless)
   scan/parse error, line 3: syntax error at: unless
   .sIdentifier
   ?pColon (pUnless)
   @TypeBody
...
```

Constants

Valid Declaration

First we used `constants_valid.pt` to test if the parser outputted the correct semantic tokens for declaring constants in Quby. The file contains two constant declarations and a `using` statement to complete the program.

The expected output of the program is (Note the double slash `//` refers to our own added comments to the system output):

```
.sProgram        // For the pUsing token
.sIdentifier     // Identifier of the using statement
.sParmEnd        // Part of the program definition
.sBegin          // Block begin statement
.sConst          // the constant semantic token, triggered by pVal token
.sIdentifier     // identifer for "a"
.sInteger        // the constant assigned to "a"
.sConst          // the constant semantic token, triggered by pVal token
.sIdentifier     // identifier for "b"
.sStringLiteral  // The string constant assigned to "b"
.sEnd            // Block end statement
```

By running `parsetrace` on the file, we get the following output:

```
 % .sNewLine
 .sProgram
 % .sNewLine
 .sIdentifier
 .sParmEnd
  .sBegin
  .sConst
   .sIdentifier
    % .sNewLine
    .sInteger
  .sConst
```

```
    .sIdentifier
     .sStringLiteral
   .sEnd
```

This matches our expected output, with the only change being the addition of the `% .sNewLine`, which are automatically emitted by the parser. This shows that the parser can correctly parse constant assignment statements in Quby.

Invalid Declaration

We also test invalid constant declarations to ensure the parser throws an error when they are seen. Consider `constants_invalid_1.pt` where the constant assignment is done with a colon rather than an `=`. While the emitted tokens look the same:

```
% .sNewLine
 .sProgram
 % .sNewLine
 .sIdentifier
 .sParmEnd
  .sBegin
  .sConst
   .sIdentifier
    .sInteger
  .sEnd
```

Looking at the entire parsertrace output, we see that an error is raised when we encounter the token. This will be caught by the full compiler:

```
...
  @ConstantDefinitions
   ?pIdentifier (pIdentifier)
   .sIdentifier
   ?pAssignEquals (pColon)
   scan/parse error, line 3: syntax error at: :
   @ConstantValue
    [ (pColon)
    | *:
    ?pInteger (pColon)
    }
...
```

We also try declaring multiple constants at once using the comma separated notation only reserved for variable declaration. This is done in `constants_invalid_2.pt` whose output is:

```
  % .sNewLine
  .sProgram
  % .sNewLine
  .sIdentifier
  .sParmEnd
   .sBegin
   .sConst
    .sIdentifier
     .sInteger
   .sEnd
```

The outputted semantic tokens only recognize one constant declaration instead of the others. Furthermore, a parser error is thrown due to the occurrence of the comma:

```
...
    .sIdentifier
    ?pAssignEquals (pComma)
    scan/parse error, line 3: syntax error at: ,
    @ConstantValue
     [ (pComma)
     | *:
     ?pInteger (pComma)
     }
     .sInteger
...
```

Identifier Constant Assignment

The file `constant_idens.pt` tests constant assignment with a variable that was not previously defined. The parser trace output for this file is:

```
  % .sNewLine
  .sProgram
  % .sNewLine
  .sIdentifier
  .sParmEnd
   .sBegin
   .sConst
    .sIdentifier
     .sIdentifier
   .sEnd
```

The parser does not throw any errors as an identifier is considered a legal constant value. The parser is not able to recognize the identifier as undeclared because it performs context free syntax checking and therefore cannot see the scope of the identifier.

Type Declarations

Valid Assignments

The file `type_valid.pt` contains two valid type declarations and is the test file to show that the parser correctly parses type declarations. The expected output of the parser trace on this file is:

```
.sProgram        // For the pUsing token
.sIdentifier     // Identifier of the using statement
.sParmEnd        // Part of the program definition
.sBegin          // Block begin statement

.sType       // type semantic token
.sIdentifier // type identifier for "t"
.sIdentifier // type assigned for t
.sType       // type semantic token
.sIdentifier // type identifier for "p"
.sIdentifier // type assigned for p

.sEnd            // Block end statement
```

This is validated by the actual parsertrace output:

```
 % .sNewLine
 .sProgram
 % .sNewLine
 .sIdentifier
 .sParmEnd
  .sBegin
  .sType
   .sIdentifier
     % .sNewLine
     .sIdentifier
  .sType
   .sIdentifier
     .sIdentifier
  .sEnd
```

Since a `pIdentifier` is considered a valid type (as shown in SimpleType), undeclared user types will not cause parser errors as that requires context aware checking (the parser is nt aware of the scope of the identifiers).

Invalid Assignments

The file `type_invalid_1.pt` attempts to create a type with the `=` operator rather than the `:` operator. While the parser trace output indicates no issues:

```
 % .sNewLine
 .sProgram
 % .sNewLine
 .sIdentifier
 .sParmEnd
  .sBegin
  .sType
   .sIdentifier
       .sIdentifier
     .sRange
       .sIdentifier
  .sEnd
```

Investigating the full parser trace shows that a parsing error was thrown when the parser expected a `:` in TypeBody but instead got the `=`:

```
...
  @TypeDefinitions
   ?pIdentifier (pIdentifier)
   .sIdentifier
   ?pColon (pAssignEquals)
   scan/parse error, line 3: syntax error at: =
   @TypeBody
    [ (pAssignEquals)
    | *:
    @SimpleType
     [ (pAssignEquals)
...
```

Variables

Valid Assignments

The file `variables_valid.pt` is designed to test if the parser correctly parses a variable declarations. It contains two one-variable declarations on separate lines. The expected output is:

```
.sProgram        // For the pUsing token
.sIdentifier     // Identifier of the using statement
.sParmEnd        // Part of the program definition
.sBegin          // Block begin statement

.sVar         // variable declaration token
```

```
.sIdentifier  // identifier for variable to be declared: "a"
.sIdentifier  // identifier for variable type of a
.sVar         // variable declaration token
.sIdentifier  // identifier for variable to be declared: "b"
.sIdentifier  // identifier for variable type of b

.sEnd            // Block end statement
```

This is verified by the actual parser trace output shown below:

```
 % .sNewLine
 .sProgram
 % .sNewLine
 .sIdentifier
 .sParmEnd
  .sBegin
  .sVar
   .sIdentifier
     % .sNewLine
     .sIdentifier
  .sVar
   .sIdentifier
     .sIdentifier
  .sEnd
```

We also have the test file `variables_commas_valid.pt` to test the parsing of multi-variable declarations using the comma separation that is introduced in Quby. This file is a modifications of `variables_valid.pt`, where instead we have one two-variable declaration.

The expected output for this case would be the variables token and the identifier token, ending with the type to be assigned to the variables:

```
.sProgram        // For the pUsing token
.sIdentifier     // Identifier of the using statement
.sParmEnd        // Part of the program definition
.sBegin          // Block begin statement

.sVar         // variable declaration token
.sIdentifier  // identifier for variable to be declared: "a"
.sVar         // variable declaration token
.sIdentifier  // identifier for variable to be declared: "b"
.sIdentifier  // identifier for variable type of b and a

.sEnd            // Block end statement
```

This is verified by the actual parser trace output shown below:

```
 % .sNewLine
 .sProgram
 % .sNewLine
 .sIdentifier
 .sParmEnd
  .sBegin
  .sVar
   .sIdentifier
   .sVar
   .sIdentifier
     .sIdentifier
  .sEnd
```

Invalid Assignments

The file `variables_invalid_1.pt` tries to declare variables with the `=` operator rather than the expected `colon` operator. As the case with types, the parser throws an error as it was expecting the colon:

```
...
   ?pColon (pAssignEquals)
   scan/parse error, line 3: syntax error at: =
   @TypeBody
    [ (pAssignEquals)
    | *:
    @SimpleType
     [ (pAssignEquals)
...
```

We also have `variables_invalid_2.pt` which misuses the comma notation for multi-variables. It uses the comma but does not specify a variable name.

The parser throws an error for this, because (as defined the VariableDeclarations rule) the parser is expecting a `pIdentifier` if a `,` is specified after the first variable identifier:

```
...
   [ (pComma)
   | pComma:
   .sVar
   ?pIdentifier (pColon)
   scan/parse error, line 3: syntax error at: :
   .sIdentifier
   ] or >
```

```
    }
...
```

Ethan's Testing

Testing is split into three sections, the testing for procedures, that for modules, and the final section which deals with program parsing more broadly.

All test files can be found in ptsrc/test/phase-2/ethan.

Procedures

While the general logic for the procedures in Quby is similar to PT Pascal, changes were made to syntax and functionality, so testing is necessary.

Public

The Phase 2 specifications state that the sPublic token must be emitted after the procedure sIdentifier. Tests for public procedures with and without parameters are shown below.

Without Parameters

The test file used to generate the following output is *public_def.pt*.

```
 .sProgram
 % .sNewLine
 % .sNewLine
 .sIdentifier
 .sParmEnd
  .sBegin
  .sProcedure
  .sIdentifier
  .sPublic
   .sIdentifier
   .sIdentifier
   .sParmEnd
   .sBegin
   .sEnd
  .sEnd
```

As shown above, the sPublic token is emitted successfully and in the right place.

With Parameters

The test file used to generate the following output is *public_def_with_params.pt*.

```
.sProgram
% .sNewLine
% .sNewLine
.sIdentifier
.sParmEnd
 .sBegin
 .sProcedure
 .sIdentifier
 .sPublic
  .sIdentifier
  .sIdentifier
  .sIdentifier
  .sIdentifier
  .sIdentifier
  .sVar
  .sIdentifier
  % .sNewLine
  .sParmEnd
  .sBegin
  .sVar
   .sIdentifier
     % .sNewLine
     .sIdentifier
   .sAssignmentStmt
   .sIdentifier
        % .sNewLine
        .sIdentifier
   .sExpnEnd
  % .sNewLine
  .sEnd
 .sEnd
```

As shown above, the sPublic token is emitted successfully and in the right place. In addition, all parameters are identified successfully and in the right order. The begin and end tokens appear in their appropriate locations, and statements within procedures are recognized.

Private

The private procedures are identical to public procedures less the sPublic token, and are processed identically as is shown below.

Without Parameters

The test file used to generate the following output is *private_def.pt*.

```
.sProgram
% .sNewLine
% .sNewLine
.sIdentifier
.sParmEnd
 .sBegin
 .sProcedure
 .sIdentifier
  .sIdentifier
  .sIdentifier
  .sParmEnd
  .sBegin
  .sEnd
 .sEnd
```

With Parameters

The test file used to generate the following output is *private_def_with_params.pt*.

```
.sProgram
% .sNewLine
% .sNewLine
.sIdentifier
.sParmEnd
 .sBegin
 .sProcedure
 .sIdentifier
  .sIdentifier
  .sIdentifier
  .sIdentifier
  .sIdentifier
  .sIdentifier
  .sVar
  .sIdentifier
  % .sNewLine
  .sParmEnd
  .sBegin
  .sVar
   .sIdentifier
     % .sNewLine
     .sIdentifier
   .sAssignmentStmt
   .sIdentifier
        % .sNewLine
```

```
          .sIdentifier
     .sExpnEnd
    % .sNewLine
    .sEnd
   .sEnd
```

Modules

Primary testing and design of the module was carried out by Liam, so the testing in this segment is limited to the functionality of the sPublic token, which will be shown below.

Public

Modules can only be accessed when declared public in Quby, so ensuring sPublic tokens are emitted is essential.

The test file used to generate the following output is *public_module.pt*.

```
  .sProgram
  % .sNewLine
  % .sNewLine
  .sIdentifier
  .sParmEnd
   .sBegin
    .sModule
    % .sNewLine
    .sIdentifier
    .sPublic
     .sBegin
     .sVar
      .sIdentifier
        % .sNewLine
        .sIdentifier
     % .sNewLine
     .sEnd
   .sEnd
```

As shown above, both the sPublic and sModule tokens are emitted correctly and in proper order.

Private

Testing of private modules can be seen in Liam's testing section

Routine Recognition

This section pertains to the correct parsing of a complete and correct Quby program containing multiple procedures and/or modules with variables and parameters. The specified procedures/modules have varied visibility to make the test more realistic, though the primary aim is to ensure procedures and modules are recognized more broadly, which is shown below.

The test file used to generate the following output is *full_routine.pt*.

```
.sProgram
% .sNewLine
% .sNewLine
.sIdentifier
.sParmEnd
 .sBegin
 .sConst
  .sIdentifier
   % .sNewLine
   .sInteger
 .sConst
  .sIdentifier
   % .sNewLine
   % .sNewLine
   .sInteger
 .sProcedure
 .sIdentifier
  .sIdentifier
  .sIdentifier
  .sIdentifier
  .sIdentifier
  .sIdentifier
  .sVar
  .sIdentifier
  % .sNewLine
  .sParmEnd
  .sBegin
  .sVar
   .sIdentifier
     % .sNewLine
     .sIdentifier
   .sAssignmentStmt
   .sIdentifier
        % .sNewLine
        .sIdentifier
   .sExpnEnd
  % .sNewLine
  % .sNewLine
```

```
    .sProcedure
    % .sNewLine
    .sIdentifier
    .sPublic
     .sParmEnd
     .sBegin
     .sVar
      .sIdentifier
        % .sNewLine
        .sIdentifier
     % .sNewLine
     % .sNewLine
      .sModule
      % .sNewLine
      .sIdentifier
      .sPublic
       .sBegin
       .sVar
        .sIdentifier
          % .sNewLine
          .sIdentifier
       % .sNewLine
       .sEnd
     .sEnd
    .sEnd
   .sEnd
```

Noah's Testing

All test files mentioned in this file are located in the folder 'ptsrc/test/phase-2/noah'.

Unless Statement

Valid Example

A valid example of the use of the Quby unless statement can be found in the file 'unless.pt'. The output is transformed as expected into an if statement with its expression notted. As well, the body of the statement is bounded by an `.sBegin` and an `.sEnd` token.

```
.sProgram
 % .sNewLine
 .sIdentifier
 .sParmEnd
  .sBegin
   .sIfStmt
```

```
          .sIdentifier
          .sInteger
      .sEq
     .sNot
     .sExpnEnd
     % .sNewLine
     .sThen
      .sBegin
       .sAssignmentStmt
       .sIdentifier
            % .sNewLine
            .sIdentifier
       .sExpnEnd
      .sEnd
    .sEnd
```

Invalid Example

The unless statement in Quby is required to have the `then` keyword token after its condition, the omission of the `then`, as seen in 'bad_unless.pt' makes the compiler fail as shown below in the following output.

```
...
    >>
   ;Expression
   .sNot
   .sExpnEnd
   ?pThen (pIdentifier)
   scan/parse error, line 4: syntax error at: y
   .sThen
   @Block
    .sBegin
    [ (pIdentifier)
    | *:
    ] or >
    .sEnd
...
```

Case statement

Valid Examples

The case statement can appear in two general forms in Quby: with a default clause or without one. The first valid testing file, 'case_1.pt' shows the proper output from a case statement with a default `else` clause:

```
.sProgram
% .sNewLine
.sIdentifier
.sParmEnd
 .sBegin
  .sCaseStmt
       % .sNewLine
       .sIdentifier
  .sExpnEnd
     .sInteger
   .sLabelEnd
   % .sNewLine
    .sBegin
     .sAssignmentStmt
     .sIdentifier
          % .sNewLine
          .sInteger
     .sExpnEnd
    .sEnd
     .sInteger
   .sLabelEnd
   % .sNewLine
    .sBegin
     .sAssignmentStmt
     .sIdentifier
          % .sNewLine
          .sInteger
     .sExpnEnd
    .sEnd
  % .sNewLine
  .sElse
   .sBegin
    .sAssignmentStmt
    .sIdentifier
         % .sNewLine
         .sInteger
    .sExpnEnd
   .sEnd
  .sCaseEnd
 .sEnd
```

The second testing file, 'case_2.pt', is the same as the first, except it removes the optional else clause from the case statement. Its output can be seen below:

```
.sProgram
% .sNewLine
.sIdentifier
.sParmEnd
 .sBegin
  .sCaseStmt
       % .sNewLine
       .sIdentifier
  .sExpnEnd
     .sInteger
   .sLabelEnd
   % .sNewLine
    .sBegin
     .sAssignmentStmt
     .sIdentifier
          % .sNewLine
          .sInteger
     .sExpnEnd
    .sEnd
     .sInteger
   .sLabelEnd
   % .sNewLine
    .sBegin
     .sAssignmentStmt
     .sIdentifier
          % .sNewLine
          .sInteger
     .sExpnEnd
    .sEnd
  .sCaseEnd
 .sEnd
```

Invalid Examples

There are three test files for invalid case statements. The first, 'bad_case_1.pt' tests the previous PT Pascal case statement syntax and as expected, it errors when encountering the previous `of` keyword:

```
...
    ;SimpleExpression
    [ (pOf)
    | *:
    >>
   ;Expression
```

```
    .sExpnEnd
    ?pWhen (pOf)
    scan/parse error, line 3: syntax error at: of
    @CaseAlternative
...
```

The second test file, 'bad_case_2.pt' tests a case statement that has a `when` clause without a matching `then`.

```
...
     ;OptionallySignedIntegerConstant
     [ (pIdentifier)
     | *:
     ] or >
     .sLabelEnd
     ?pThen (pIdentifier)
     scan/parse error, line 5: syntax error at: y
     @Block
      .sBegin
      [ (pIdentifier)
      | *:
      ] or >
      .sEnd
      >>
     ;Block
...
```

And thirdly, the last test file, 'bad_case_3.pt' tests a case statement that does not have any `when` clauses with a sole `else` statement, which also fails since it is looking for a `.sWhen` token when one is not to be found.

```
...
     ;SimpleExpression
     [ (pElse)
     | *:
     >>
    ;Expression
    .sExpnEnd
    ?pWhen (pElse)
    scan/parse error, line 4: syntax error at: else
    @CaseAlternative
     @OptionallySignedIntegerConstant
      [ (pElse)
      | *:
```

```
    @UnsignedIntegerConstant
...
```

Do Statement

Valid Examples

The do statement in Quby must start with the keyword `do` and end with the keyword `end` and in its body must contain one or more `break if` statement. The testing file 'do_1.pt' adheres to the proper syntax and as expected produces the following valid token output stream:

```
 .sProgram
 % .sNewLine
 .sIdentifier
 .sParmEnd
  .sBegin
  % .sNewLine
   .sDo
   .sBegin
    .sBegin
     .sAssignmentStmt
     .sIdentifier
          % .sNewLine
          .sInteger
     .sExpnEnd
    .sEnd
   .sBreakIf
        .sIdentifier
        % .sNewLine
        .sInteger
    .sEq
   .sExpnEnd
    .sBegin
     .sAssignmentStmt
     .sIdentifier
          % .sNewLine
          .sInteger
     .sExpnEnd
    .sEnd
   .sEnd
  .sEnd
```

As well, the do statement has the option to include more than one `break if` statement. This behaviour is testing in the file 'do_2.pt', and as expected, it parses successfully with the following output:

```
% .sNewLine
.sProgram
% .sNewLine
.sIdentifier
.sParmEnd
.sBegin
% .sNewLine
 .sDo
 .sBegin
  .sBegin
   .sAssignmentStmt
   .sIdentifier
        % .sNewLine
        .sInteger
   .sExpnEnd
  .sEnd
 .sBreakIf
      .sIdentifier
      % .sNewLine
      .sInteger
  .sEq
 .sExpnEnd
  .sBegin
   .sAssignmentStmt
   .sIdentifier
        % .sNewLine
        .sInteger
   .sExpnEnd
  .sEnd
 .sBreakIf
      .sIdentifier
      % .sNewLine
      .sInteger
  .sEq
 .sExpnEnd
 .sBreakIf
      .sIdentifier
      % .sNewLine
      .sInteger
  .sEq
 .sExpnEnd
 .sEnd
.sEnd
```

Invalid Example

The do loop is coming into the Quby language as a form of replacing the repeat loop and thus the repeat loop syntax has been deprecated. In the file 'bad_repeat.pt', the old syntax is utilized and as expected the program fails.

```
...
   | *:
   @CallStmt
    .sCallStmt
    .sIdentifier
    [ (pEquals)
    | *:
    .sParmEnd
    >>
   ;CallStmt
   >>
  ;AssignmentOrCallStmt
  ] or >
  }
  [ (pEquals)
  | *:
  ] or >
  .sEnd
  >>
 ;Block
 >>
;Program
scan/parse error, line 5: syntax error at: =
```

To test the new do syntax more rigorously, an additional test file is provided, 'bad_do.pt'. In this file a do statement is tested without the required `break if`, and as expected, it fails.

```
...
    [ (pEnd)
    | *:
    ] or >
    .sEnd
    >>
   ;Block
   ?pBreak (pEnd)
   scan/parse error, line 6: syntax error at: end
   ?pIf (pEnd)
   .sBreakIf
   @Expression
```

```
    @SimpleExpression
     [ (pEnd)
     | *:
     @Term
...
```

Liam's Testing

All testing documentation used can be found in /ptsrc/test/phase-2/liam/

`module` testing

The code module.pt in the /ptsrc/test/phase-2/liam/ directory outputted the correct code shown below. This test case was used to see a correct implementation of the `module` declaration. As seen below, after the `.sModule` the preceding token that is emitted correlate to the subsequent identifier that is used for identifying the module. Then, all subsequent declarations are encapsulated with the `.sBegin` and `.sEnd` generated by the block rule being called.

```
 .sProgram
 .sIdentifier
 .sParmEnd
  .sBegin
  % .sNewLine
  % .sNewLine
   .sModule
   % .sNewLine
   .sIdentifier
    .sBegin
     .sAssignmentStmt
     .sIdentifier
          % .sNewLine
          .sInteger
     .sExpnEnd
     .sAssignmentStmt
     .sIdentifier
          % .sNewLine
          .sInteger
     .sExpnEnd
    .sEnd
  .sEnd
```

Erroneous test case for `module`:

In the module error pt file, module_error.pt, the modules identifier is removed. An error is thrown at the subsequent line were x = 1 is. Below is the following output:

```
@Program
 ?pUsing (pUsing)
 .sProgram
 ?pIdentifier (pIdentifier)
 .sIdentifier
 [ (pSemicolon)
 | *:
 ] or >
 .sParmEnd
 @Block
  .sBegin
  [ (pSemicolon)
  | pSemicolon:
  % .sNewLine
  % .sNewLine
  ] or >
  }
  [ (pModule)
  | pModule:
  % .sNewLine
  @Module
   .sModule
   ?pIdentifier (pIdentifier)
   .sIdentifier
   @Block
    .sBegin
    [ (pAssignEquals)
    | *:
    ] or >
    .sEnd
    >>
   ;Block
   >>
  ;Module
  ] or >
  }
  [ (pAssignEquals)
  | *:
  ] or >
  .sEnd
  >>
 ;Block
 >>
```

```
;Program
scan/parse error, line 4: syntax error at: =
```

`while` testing

The code from while.pt outputted the following. This code correctly outputted the sBegin and sEnd to encompass the statements encapsulated by the loop. This test case is correct as we can the `.sWhileStmt` being emitted and not a `.sDo`. Then a subsequent expression is emitted. Then the rest of the declarations that exist inside the loop are emitted and encapsulated in the `.sBegin` and `.sEnd` from the block rule.

```
 .sProgram
 .sIdentifier
 .sParmEnd
  .sBegin
  % .sNewLine
  % .sNewLine
   .sWhileStmt
        .sIdentifier
        .sInteger
    .sEq
   .sExpnEnd
   % .sNewLine
    .sBegin
     .sAssignmentStmt
     .sIdentifier
          .sIdentifier
          % .sNewLine
          .sInteger
       .sAdd
     .sExpnEnd
    .sEnd
  .sEnd
```

Erroneus test case for `while` loops:

In this test case the while loop is missing the subsequent do after the expression. The file while_error.pt had the following output:

```
@Program
 ?pUsing (pUsing)
 .sProgram
 ?pIdentifier (pIdentifier)
 .sIdentifier
 [ (pSemicolon)
```

```
| *:
] or >
.sParmEnd
@Block
 .sBegin
 [ (pSemicolon)
 | pSemicolon:
 % .sNewLine
 % .sNewLine
 ] or >
 }
 [ (pWhile)
 | pWhile:
 @WhileStmt
  .sWhileStmt
  @Expression
   @SimpleExpression
    [ (pIdentifier)
    | *:
    @Term
     @Subterm
      [ (pIdentifier)
      | *:
      @Factor
       [ (pIdentifier)
       | pIdentifier:
       .sIdentifier
       @IdentifierExtension
        [ (pEquals)
        | *:
        >>
       ;IdentifierExtension
       ] or >
       >>
      ;Factor
      >>
     ;Subterm
     [ (pEquals)
     | *:
     ] or >
     >>
    ;Term
    [ (pEquals)
    | *:
```

```
    ] or >
    >>
   ;SimpleExpression
   [ (pEquals)
   | pEquals:
   @SimpleExpression
    [ (pInteger)
    | *:
    @Term
     @Subterm
      [ (pInteger)
      | *:
      @Factor
       [ (pInteger)
       | pInteger:
       % .sNewLine
       .sInteger
       ] or >
       >>
      ;Factor
      >>
     ;Subterm
     [ (pIdentifier)
     | *:
     ] or >
     >>
    ;Term
    [ (pIdentifier)
    | *:
    ] or >
    >>
   ;SimpleExpression
   .sEq
   ] or >
   >>
  ;Expression
  .sExpnEnd
  ?pDo (pIdentifier)
  scan/parse error, line 4: syntax error at: x
  @Block
   .sBegin
   [ (pIdentifier)
   | *:
   ] or >
```

```
      .sEnd
      >>
     ;Block
     >>
    ;WhileStmt
    ] or >
    }
    [ (pIdentifier)
    | *:
    ] or >
    .sEnd
    >>
   ;Block
   >>
  ;Program
```

`if` testing

The pt code used to test the if statements can be found in '/ptsrc/test/phase-2/liam/if.pt'. This code was used to test the correct use case of if statements.

The testing code outputted the following below. The reason this output is correct is because we first emit the `.sIfStmt` and then emit the subsequent expression correlating to the if. Then, the `.sThen` is emitted to signify that all subsequent code within the `.sBegin` and `.sEnd` correlate to the if statement.

```
   .sProgram
   .sIdentifier
   .sParmEnd
    .sBegin
    % .sNewLine
    % .sNewLine
     .sIfStmt
          .sIdentifier
          .sInteger
      .sEq
     .sExpnEnd
     % .sNewLine
     .sThen
      .sBegin
       .sAssignmentStmt
       .sIdentifier
            .sInteger
       .sExpnEnd
      % .sNewLine
```

```
    .sEnd
  .sEnd
```

Erroneus test case for `if` statements:

In the file if_error.pt, there is a missing then after an expression. The code had the following output:

```
@Program
 ?pUsing (pUsing)
 .sProgram
 ?pIdentifier (pIdentifier)
 .sIdentifier
 [ (pSemicolon)
 | *:
 ] or >
 .sParmEnd
 @Block
  .sBegin
  [ (pSemicolon)
  | pSemicolon:
  % .sNewLine
  % .sNewLine
  ] or >
  }
  [ (pIf)
  | pIf:
  @IfStmt
   .sIfStmt
   @Expression
    @SimpleExpression
     [ (pIdentifier)
     | *:
     @Term
      @Subterm
       [ (pIdentifier)
       | *:
       @Factor
        [ (pIdentifier)
        | pIdentifier:
        .sIdentifier
        @IdentifierExtension
         [ (pEquals)
         | *:
         >>
```

```
          ;IdentifierExtension
          ] or >
          >>
         ;Factor
         >>
        ;Subterm
        [ (pEquals)
        | *:
        ] or >
        >>
       ;Term
       [ (pEquals)
       | *:
       ] or >
       >>
      ;SimpleExpression
      [ (pEquals)
      | pEquals:
      @SimpleExpression
       [ (pInteger)
       | *:
       @Term
        @Subterm
         [ (pInteger)
         | *:
         @Factor
          [ (pInteger)
          | pInteger:
          % .sNewLine
          .sInteger
          ] or >
          >>
         ;Factor
         >>
        ;Subterm
        [ (pIdentifier)
        | *:
        ] or >
        >>
       ;Term
       [ (pIdentifier)
       | *:
       ] or >
       >>
```

```
    ;SimpleExpression
    .sEq
    ] or >
    >>
   ;Expression
   .sExpnEnd
   ?pThen (pIdentifier)
   scan/parse error, line 4: syntax error at: x
   .sThen
   @Block
    .sBegin
    [ (pIdentifier)
    | *:
    ] or >
    .sEnd
    >>
   ;Block
   [ (pIdentifier)
   | *:
   >>
  ;IfStmt
  ] or >
  }
  [ (pIdentifier)
  | *:
  ] or >
  .sEnd
  >>
 ;Block
 >>
;Program
```

`else` changes

The if_else.pt code is a correct use case of the 'else' statement and had the following output:

```
 .sProgram
 .sIdentifier
 .sParmEnd
  .sBegin
  % .sNewLine
  % .sNewLine
   .sIfStmt
        .sIdentifier
```

```
          .sInteger
      .sEq
     .sExpnEnd
     % .sNewLine
     .sThen
      .sBegin
       .sAssignmentStmt
       .sIdentifier
            % .sNewLine
            .sInteger
       .sExpnEnd
      .sEnd
     % .sNewLine
     .sElse
      .sBegin
       .sAssignmentStmt
       .sIdentifier
            % .sNewLine
            .sInteger
       .sExpnEnd
      .sEnd
    .sEnd
```

The above output signifies correct functionality as the `.sElse` being emitted is subsequent the if's block rule. Then a subsequent `.sBegin` and `.sEnd` are used to encapsulated all declerations within the else portion.

Erroneus test case for `else` statements:

The file else_error.pt tested having multiple subsequent else statements. The code had the following output error message:

```
@Program
 ?pUsing (pUsing)
 .sProgram
 ?pIdentifier (pIdentifier)
 .sIdentifier
 [ (pSemicolon)
 | *:
 ] or >
 .sParmEnd
 @Block
  .sBegin
  [ (pSemicolon)
  | pSemicolon:
```

```
    % .sNewLine
    % .sNewLine
    ] or >
    }
    [ (pElse)
    | *:
    ] or >
    .sEnd
    >>
   ;Block
   >>
  ;Program
  scan/parse error, line 3: syntax error at: else
```

`elsif` changes

The code below was used to test proper use case of the elsif statment.
The if_elsif.pt code outputted the following, placing the nested if rule within a `.sElse` `.sBegin` and a `.sEnd`:

```
   .sProgram
   .sIdentifier
   .sParmEnd
    .sBegin
    % .sNewLine
    % .sNewLine
     .sIfStmt
          .sIdentifier
          .sInteger
      .sEq
     .sExpnEnd
     % .sNewLine
     .sThen
      .sBegin
       .sAssignmentStmt
       .sIdentifier
            % .sNewLine
            .sInteger
       .sExpnEnd
      .sEnd
     .sElse
     .sBegin
      .sIfStmt
           .sIdentifier
```

```
        .sInteger
    .sEq
   .sExpnEnd
   % .sNewLine
   .sThen
    .sBegin
     .sAssignmentStmt
     .sIdentifier
          % .sNewLine
          .sInteger
     .sExpnEnd
    .sEnd
  .sEnd
 .sEnd
```

The above output shows that the elsif rule is functioning properly, as the elsif is being emitted as a nested `.sIfStmt` within the first if statments `.sElse`. Even though the pt code had no `else` within the code, it was still emitted to allow the `elsif` to be treated as a nested `if` within the `else`.

Erroneus Test Case for `elsif` statements:

The file elsif_error.pt tests to see if placing an elsif after an else could still function. The code had the following output:

```
@Program
 ?pUsing (pUsing)
 .sProgram
 ?pIdentifier (pIdentifier)
 .sIdentifier
 [ (pSemicolon)
 | *:
 ] or >
 .sParmEnd
 @Block
  .sBegin
  [ (pSemicolon)
  | pSemicolon:
  % .sNewLine
  % .sNewLine
  ] or >
  }
  [ (pIf)
  | pIf:
  @IfStmt
   .sIfStmt
```

```
@Expression
 @SimpleExpression
  [ (pIdentifier)
  | *:
  @Term
   @Subterm
    [ (pIdentifier)
    | *:
    @Factor
     [ (pIdentifier)
     | pIdentifier:
     .sIdentifier
     @IdentifierExtension
      [ (pEquals)
      | *:
      >>
     ;IdentifierExtension
     ] or >
     >>
    ;Factor
    >>
   ;Subterm
   [ (pEquals)
   | *:
   ] or >
   >>
  ;Term
  [ (pEquals)
  | *:
  ] or >
  >>
 ;SimpleExpression
 [ (pEquals)
 | pEquals:
 @SimpleExpression
  [ (pInteger)
  | *:
  @Term
   @Subterm
    [ (pInteger)
    | *:
    @Factor
     [ (pInteger)
     | pInteger:
```

```
     .sInteger
     ] or >
     >>
    ;Factor
    >>
   ;Subterm
   [ (pThen)
   | *:
   ] or >
   >>
  ;Term
  [ (pThen)
  | *:
  ] or >
  >>
 ;SimpleExpression
 .sEq
 ] or >
 >>
;Expression
.sExpnEnd
?pThen (pThen)
% .sNewLine
.sThen
@Block
 .sBegin
 [ (pIdentifier)
 | pIdentifier:
 @AssignmentOrCallStmt
  [ (pAssignEquals)
  | pAssignEquals:
  .sAssignmentStmt
  .sIdentifier
  @Expression
   @SimpleExpression
    [ (pInteger)
    | *:
    @Term
     @Subterm
      [ (pInteger)
      | *:
      @Factor
       [ (pInteger)
       | pInteger:
```

```
            % .sNewLine
            .sInteger
            ] or >
            >>
           ;Factor
           >>
          ;Subterm
          [ (pElse)
          | *:
          ] or >
          >>
         ;Term
         [ (pElse)
         | *:
         ] or >
         >>
        ;SimpleExpression
        [ (pElse)
        | *:
        >>
       ;Expression
       .sExpnEnd
       ] or >
       >>
      ;AssignmentOrCallStmt
      ] or >
      }
      [ (pElse)
      | *:
      ] or >
      .sEnd
      >>
     ;Block
     [ (pElse)
     | pElse:
     % .sNewLine
     .sElse
     @Block
      .sBegin
      [ (pIdentifier)
      | pIdentifier:
      @AssignmentOrCallStmt
       [ (pAssignEquals)
       | pAssignEquals:
```

```
    .sAssignmentStmt
    .sIdentifier
    @Expression
     @SimpleExpression
      [ (pInteger)
      | *:
      @Term
       @Subterm
        [ (pInteger)
        | *:
        @Factor
         [ (pInteger)
         | pInteger:
         % .sNewLine
         .sInteger
         ] or >
         >>
        ;Factor
        >>
       ;Subterm
       [ (pElsif)
       | *:
       ] or >
       >>
      ;Term
      [ (pElsif)
      | *:
      ] or >
      >>
     ;SimpleExpression
     [ (pElsif)
     | *:
     >>
    ;Expression
    .sExpnEnd
    ] or >
    >>
   ;AssignmentOrCallStmt
   ] or >
   }
   [ (pElsif)
   | *:
   ] or >
   .sEnd
```

```
     >>
    ;Block
    ] or >
    >>
   ;IfStmt
   ] or >
   }
   [ (pElsif)
   | *:
   ] or >
   .sEnd
   >>
  ;Block
  >>
 ;Program
 scan/parse error, line 7: syntax error at: elsif
```

if elsif else Case

The following code was used to test correct cases where there are subsequent `elsif`s or `else`s after an `elsif`.

The if_elsif_else.pt code tests the new implementations for if, elsif and else all together and outputted:

```
  .sProgram
  .sIdentifier
  .sParmEnd
   .sBegin
   % .sNewLine
   % .sNewLine
    .sIfStmt
         .sIdentifier
         .sInteger
     .sEq
    .sExpnEnd
    % .sNewLine
    .sThen
     .sBegin
      .sAssignmentStmt
      .sIdentifier
           % .sNewLine
           .sInteger
      .sExpnEnd
     .sEnd
```

```
     .sElse
     .sBegin
      .sIfStmt
           .sIdentifier
           .sInteger
       .sEq
      .sExpnEnd
      % .sNewLine
      .sThen
       .sBegin
        .sAssignmentStmt
        .sIdentifier
             % .sNewLine
             .sInteger
        .sExpnEnd
       .sEnd
      % .sNewLine
      .sElse
       .sBegin
        .sAssignmentStmt
        .sIdentifier
             % .sNewLine
             .sInteger
        .sExpnEnd
       .sEnd
     .sEnd
    .sEnd
```

In this case, we see that the if, elsif, else returns the correct input as the elsif and else is nested within the first `if` statements `else` that was emitted. The pt `else` statement is also correlated to the nested `if`, hence the code within it will be excecuted if both the `if` and `elsif` statements are not true.