Changelog Documentation

This documentation is divided into different sections for the parts each member of the group did. Each member of the group was responsible for the testing documentation for their changes.

- Iffy
  - Token Updates
  - Declarations
  - Minor Syntactic Details
  - String Operators
- Ethan
  - Updated the block rule with new rules and removed old rules
  - Removed the begin rule while maintaining the emission of begin tokens
  - Merged the statement rule into the block rule to make it more generally applicable
  - Added indication of public function token emission (created sPublic token)
  - Maintained PT-style emission of sBegin and sEnd tokens to minimise changes needed during semantic analysis
- Noah
  - Removed `repeat` statement
  - Added support for `do` statement
  - Modified `case` statement syntax
  - Added support for `unless` statement
- Liam
  - Added `module` to parser
  - `while` loop updated to work with new changes to Block rule
  - `if` statement changes
  - `else` statement changes
  - `elsif` statement changes

Changes made by Iffy

Overview

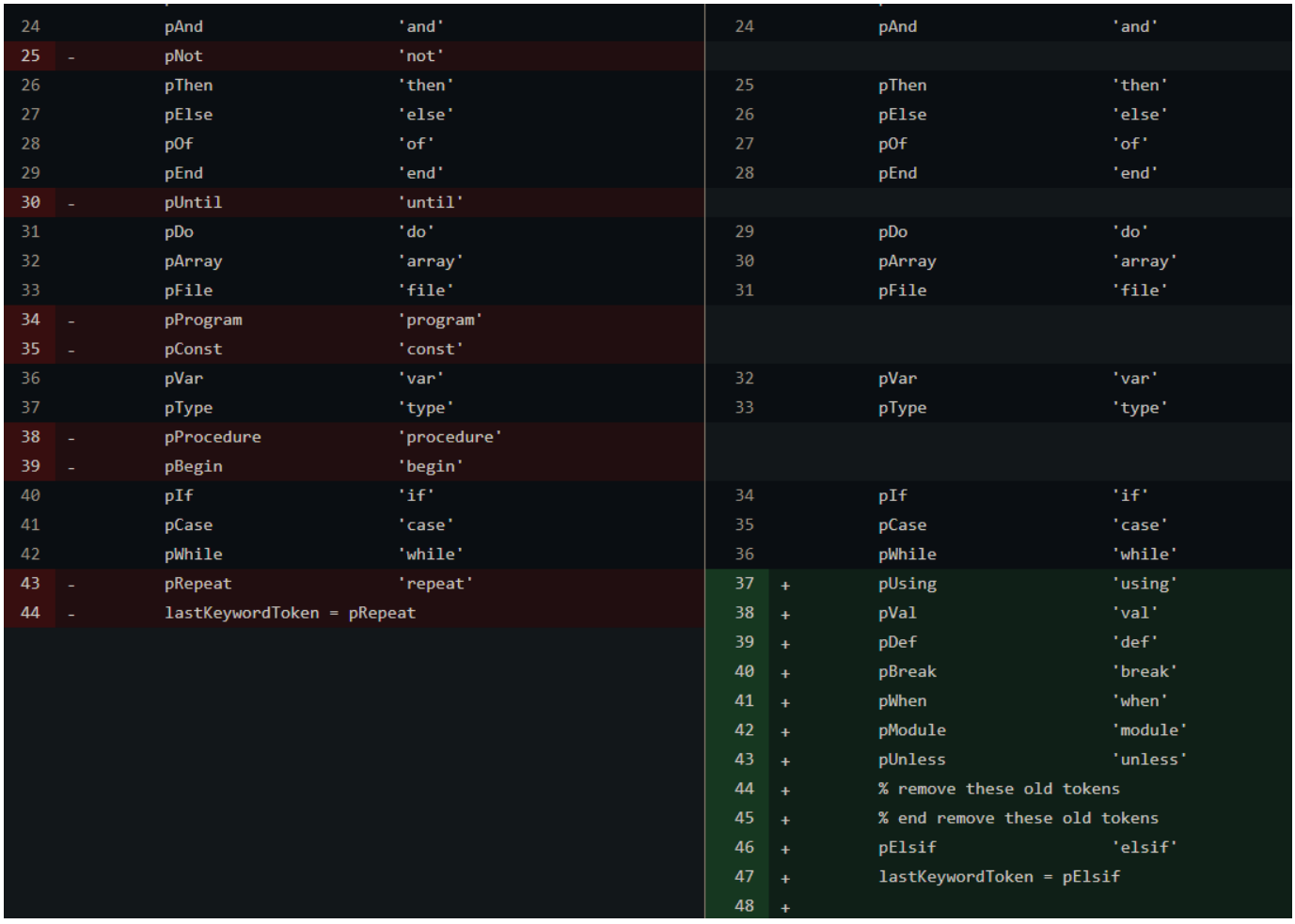
Iffy handled:

- Token Updates
- Declarations
- Minor Syntactic Details
- String Operators

Pictures were obtained using GitHub Commit Comparison.

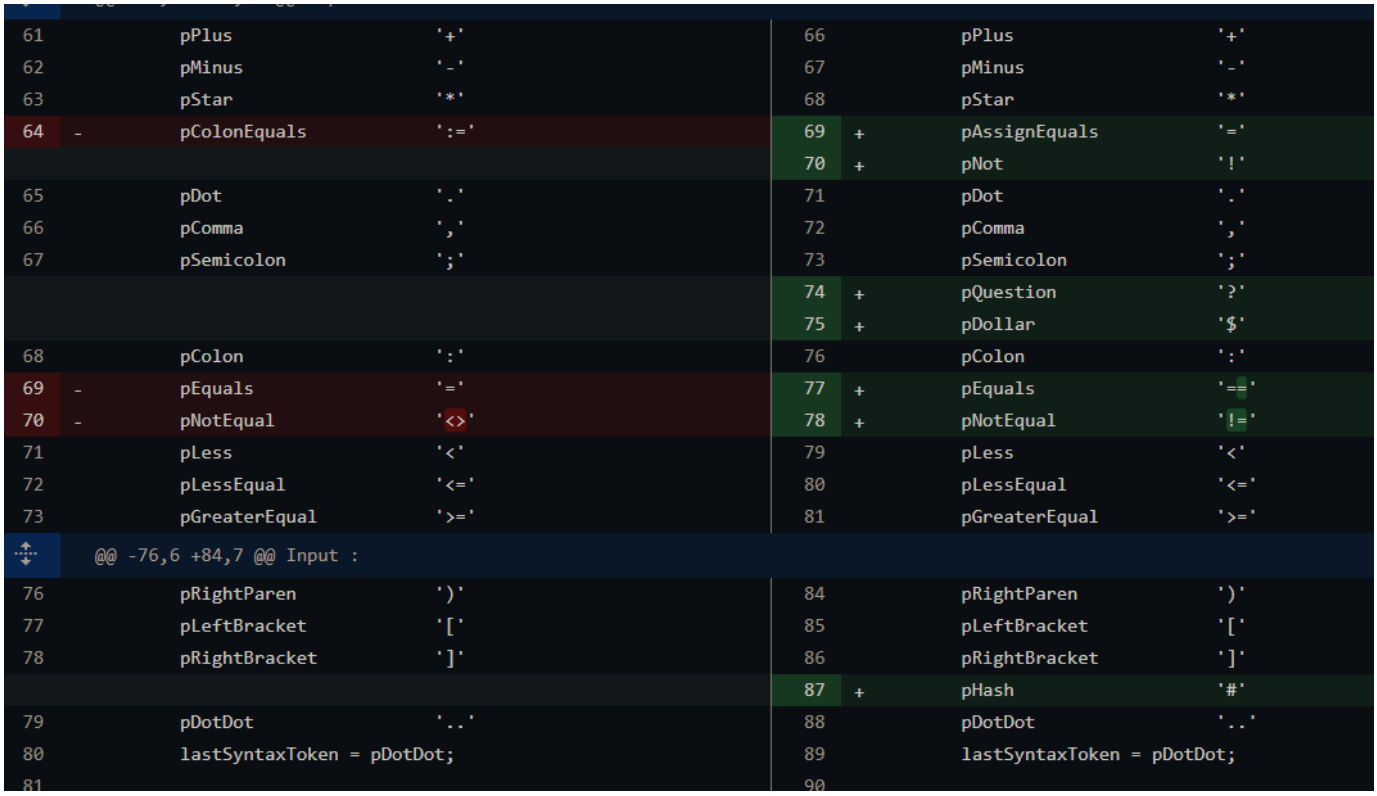
Updating Tokens

Firstly, the input tokens in parser.ssl were updated to match the output tokens in scan.ssl. This involved deleting the old unused input tokens and adding the new tokens for Quby.



```
24      pAnd                    'and'
25  -   pNot                    'not'
26      pThen                   'then'
27      pElse                   'else'
28      pOf                     'of'
29      pEnd                    'end'
30  -   pUntil                  'until'
31      pDo                     'do'
32      pArray                  'array'
33      pFile                   'file'
34  -   pProgram                'program'
35  -   pConst                  'const'
36      pVar                    'var'
37      pType                   'type'
38  -   pProcedure              'procedure'
39  -   pBegin                  'begin'
40      pIf                     'if'
41      pCase                   'case'
42      pWhile                  'while'
43  -   pRepeat                 'repeat'
44  -   lastKeywordToken = pRepeat
```

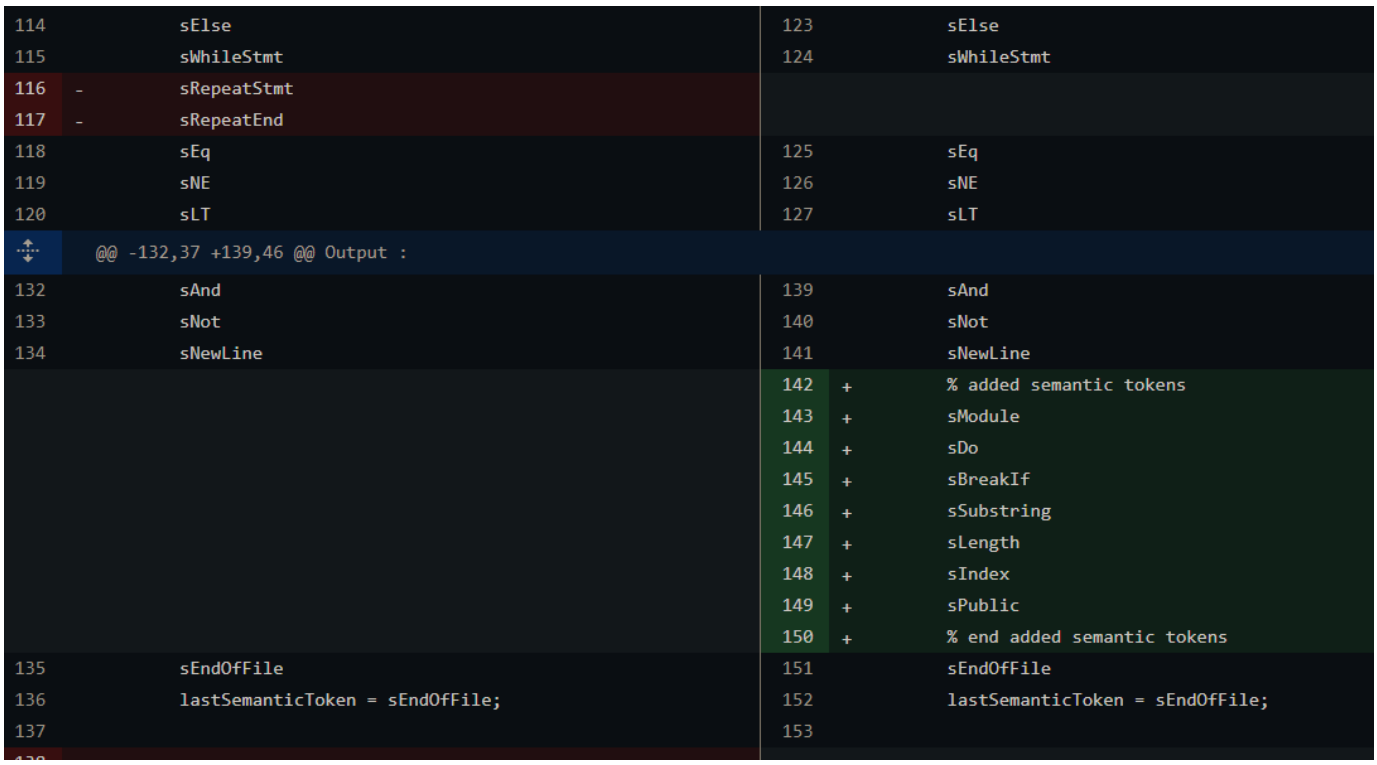
```
24      pAnd                    'and'
25      pThen                   'then'
26      pElse                   'else'
27      pOf                     'of'
28      pEnd                    'end'
29      pDo                     'do'
30      pArray                  'array'
31      pFile                   'file'
32      pVar                    'var'
33      pType                   'type'
34      pIf                     'if'
35      pCase                   'case'
36      pWhile                  'while'
37  +   pUsing                  'using'
38  +   pVal                    'val'
39  +   pDef                    'def'
40  +   pBreak                  'break'
41  +   pWhen                   'when'
42  +   pModule                 'module'
43  +   pUnless                 'unless'
44  +   % remove these old tokens
45  +   % end remove these old tokens
46  +   pElsif                  'elsif'
47  +   lastKeywordToken = pElsif
48  +
```



```
61          pPlus               '+'
62          pMinus              '-'
63          pStar               '*'
64  -       pColonEquals        ':='
65          pDot                '.'
66          pComma              ','
67          pSemicolon          ';'
68          pColon              ':'
69  -       pEquals             '='
70  -       pNotEqual           '<>'
71          pLess               '<'
72          pLessEqual          '<='
73          pGreaterEqual       '>='
    @@ -76,6 +84,7 @@ Input :
76          pRightParen         ')'
77          pLeftBracket        '['
78          pRightBracket       ']'
79          pDotDot             '..'
80          lastSyntaxToken = pDotDot;
81

66          pPlus               '+'
67          pMinus              '-'
68          pStar               '*'
69  +       pAssignEquals       '='
70  +       pNot                '!'
71          pDot                '.'
72          pComma              ','
73          pSemicolon          ';'
74  +       pQuestion           '?'
75  +       pDollar             '$'
76          pColon              ':'
77  +       pEquals             '=='
78  +       pNotEqual           '!='
79          pLess               '<'
80          pLessEqual          '<='
81          pGreaterEqual       '>='
84          pRightParen         ')'
85          pLeftBracket        '['
86          pRightBracket       ']'
87  +       pHash               '#'
88          pDotDot             '..'
89          lastSyntaxToken = pDotDot;
90
```

We also updated the semantic tokens in the system, adding the new required semantic tokens and removing the old unused semantic tokens:



```
114         sElse
115         sWhileStmt
116 -       sRepeatStmt
117 -       sRepeatEnd
118         sEq
119         sNE
120         sLT
    @@ -132,37 +139,46 @@ Output :
132         sAnd
133         sNot
134         sNewLine
135         sEndOfFile
136         lastSemanticToken = sEndOfFile;
137

123         sElse
124         sWhileStmt
125         sEq
126         sNE
127         sLT
139         sAnd
140         sNot
141         sNewLine
142 +       % added semantic tokens
143 +       sModule
144 +       sDo
145 +       sBreakIf
146 +       sSubstring
147 +       sLength
148 +       sIndex
149 +       sPublic
150 +       % end added semantic tokens
151         sEndOfFile
152         lastSemanticToken = sEndOfFile;
153
```

The semantic tokens that were added were for new operational futures introduced by Quby, this includes:

- The modules
- The `do` statement with `break if`

- The string operations: substring, length and index
- The public keyword

Declarations

Constants

Removed the semicolon ending token in the `ConstantDefinitions` rule as the semicolon ending token is not required in Quby.

Also removed the input choice loop to parse sequential constant declarations. Since Quby does not distinguish between declarations and statements, we can just have multiple constants handle by the main Block rule.

```
185
186     ConstantDefinitions :
187 -           % Accept one or more named constant
    definitions
188             pIdentifier  .sIdentifier
189 -           '=' @ConstantValue ';'
190 -           {[
191 -               | pIdentifier:
192 -                   .sIdentifier
193 -                   '=' @ConstantValue ';'
194 -               | *:
195 -                   >
196 -           ]};
197 -
```

```
223
224     ConstantDefinitions :
225 +           % Accept one named constant definitions
226             pIdentifier  .sIdentifier
227 +           '=' @ConstantValue
228 +           ;
229 +           % removed semicolon ending token
    requirement
230 +           % only has support for one constant
    definition so other calls are removed
231 +
```

Types

Removed ending semicolon requirement from `TypeDefinitions` rule and also removed the parsing of multiple type declarations (for the same reason as removing the one in `ConstantDefinitions`).

```
221     TypeDefinitions :
222             % Accept one or more named type
    definitions.
223             pIdentifier  .sIdentifier
224 -           '='  @TypeBody  ';'
225 -           {[
226 -               | pIdentifier:
227 -                   .sIdentifier
228 -                   '='  @TypeBody  ';'
229 -               | *:
230 -                   >
231 -           ]};
```

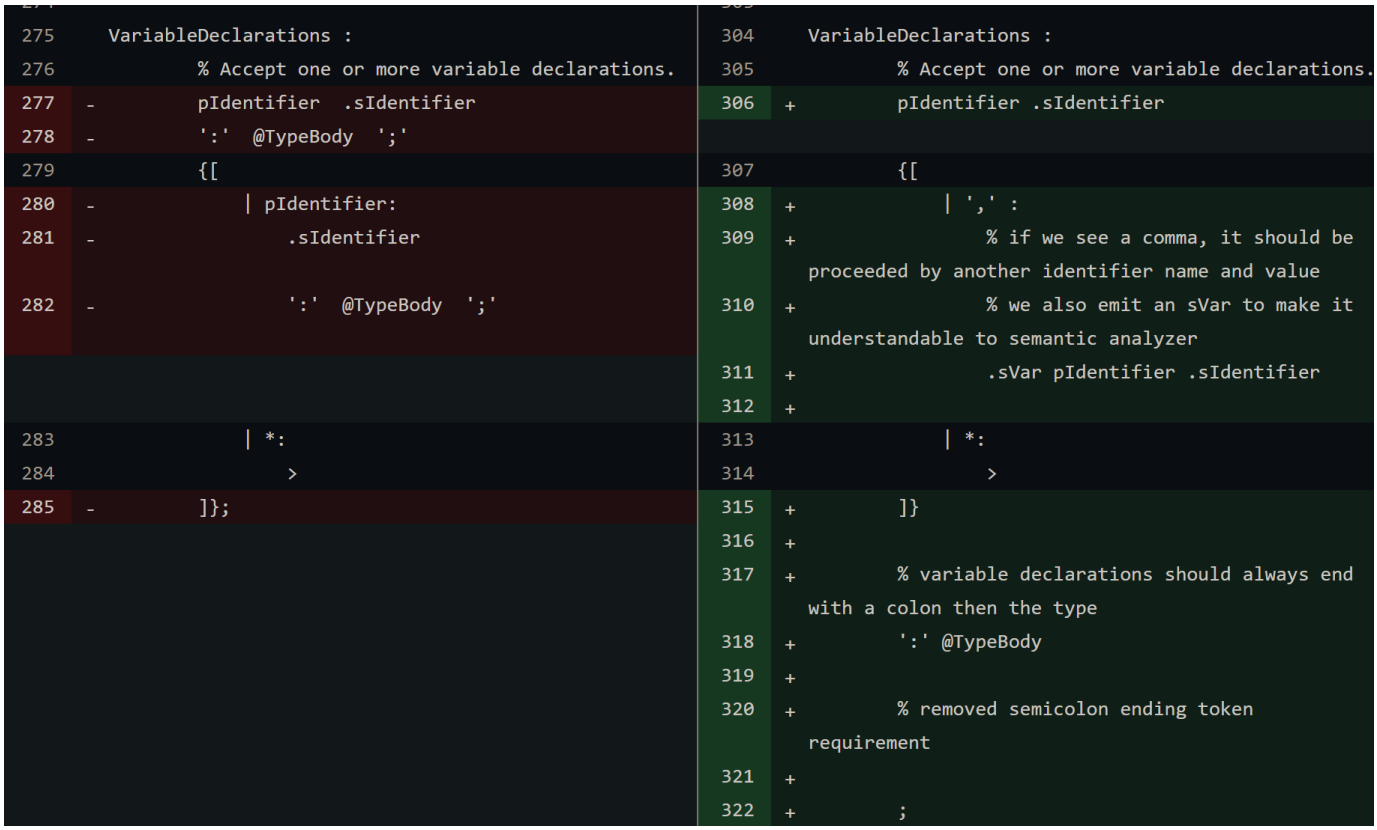
```
255     TypeDefinitions :
256             % Accept one or more named type
    definitions.
257             pIdentifier  .sIdentifier
258 +           ':'  @TypeBody
259 +           % removed semicolon ending token
    requirement
260 +           ;
```

Variables

Similar to the last two, in `VariableDeclarations`:

- Removed the ending semicolon requirement
- Removed the parsing of multiple variable declarations since that is handled by the Block rule

Also added an input choice loop to parse one-line variable declarations done with the comma.



```
275   VariableDeclarations :
276           % Accept one or more variable declarations.
277 -         pIdentifier  .sIdentifier
278 -         ':'  @TypeBody  ';'
279           {[
280 -             | pIdentifier:
281 -                 .sIdentifier
282 -                 ':'  @TypeBody  ';'
283                 | *:
284                     >
285 -         ]};
```

```
304   VariableDeclarations :
305           % Accept one or more variable declarations.
306 +         pIdentifier .sIdentifier
307           {[
308 +             | ',' :
309 +                 % if we see a comma, it should be proceeded by another identifier name and value
310 +                 % we also emit an sVar to make it understandable to semantic analyzer
311 +                 .sVar pIdentifier .sIdentifier
312 +
313                 | *:
314                     >
315 +         ]}
316 +
317 +         % variable declarations should always end with a colon then the type
318 +         ':' @TypeBody
319 +
320 +         % removed semicolon ending token requirement
321 +
322 +         ;
```

Strings

Index Operation

The string index operation `?` takes expressions as both its arguments according to the language specifications, and is at the same precedence as `div` and `mod`.

To make `?` the same precedence, it was added as a choice alternative to the Term rule, which contains the `div` and `mod` operations.

```
503     Term :
504 -       @Factor
505         {[
506             | '*':
507 -               @Factor  .sMultiply
508             | 'div':
509 -               @Factor  .sDivide
510             | 'mod':
511 -               @Factor  .sModulus
512             | 'and':
513 -               .sInfixAnd  @Factor  .sAnd
514             | *:
515                 >
516         ]};
```

```
555     Term :
556 +       @Subterm
557         {[
558             | '*':
559 +               @Subterm  .sMultiply
560             | 'div':
561 +               @Subterm  .sDivide
562             | 'mod':
563 +               @Subterm  .sModulus
564             | 'and':
565 +               .sInfixAnd  @Subterm  .sAnd
566 +           | '?':
567 +               % String index operator
568 +               @Expression
569 +               .sIndex
570             | *:
571                 >
572         ]};
```

As its choice actions, it calls the Subterm rule to maintain precedence. If the Expression rule is used instead, lower binding operators can be binded before the index operator.

We emit the `sIndex` semantic token after consuming the expression to make sure parser output is in postfix notation.

Length Operation

The string length operation `#` is also similar to `?` and takes an expression as its operand. It is at the same precedence as not, and was therefore added as a choice alternative to the Factor rule.

```
518     Factor :
519         [
520             | pIdentifier:
    @@ -524,14 +592,19 @@ Factor :
524                 .sInteger
525             | '(':
526                 @Expression  ')'
527 -           | 'not':
528                 @Factor
529                 .sNot
530             | pStringLiteral:
531                 .sStringLiteral
532             | 'file':
533                 .sFile '(' @Expression ')'
534                 .sExpnEnd
535         ];
```

```
586     Factor :
587         [
588             | pIdentifier:
592                 .sInteger
593             | '(':
594                 @Expression  ')'
595 +           | pNot: % replaced 'not' keyword
596                 @Factor
597                 .sNot
598             | pStringLiteral:
599                 .sStringLiteral
600             | 'file':
601                 .sFile '(' @Expression ')'
602                 .sExpnEnd
603 +           | '#':
604 +               % String length operand
605 +               % expecting an expression but to obey precedence rules must call expression subrules equal to or higher than its precedence
606 +               @Factor
607 +               .sLength
608         ];
```

A call to the Factor rule is done for parsing expressions again to maintain precedence. If lower binding operators are required, the contents can be surrounded by brackets which will lead to an Expression rule call in Factor.

Substring Operation

The substring operation is at a new precedence level: Higher than `div` and `mod` but lower than `not`. To implement this new precedence level, the Subterm rule was defined:

```
573
574 + Subterm:
575 +     @Factor
576 +     {[
577 +         | '$':
578 +             @Factor '..' @Factor
579 +             .sSubstring
580 +
581 +         | *:
582 +             >
583 +     ]}
584 +     ;
585 +
```

The Subterm rule is now in-between the Term and Factor rule as the new precedence level. Therefore, every call to Factor in Term was replaced with the Subterm rule (see Term rule above).

In the Subterm rule, we first make a call to Factor to process the preceding string literal that is the first operand of the substring operation. Then we have an input choice loop similar to that in Term.

If the read token is the `$`, then we call Factor to parse the range operands and then emit the `sSubstring` token to follow post fix notation. If it is anything else, we break.

Other Syntactic Details

No functional changes to the parser were required for these changes as the only thing changed were the string of characters associated with the given operation. Therefore a simple find and replace in parser.ssl was done:

- Every occurrence `<>` was replaced with the new operation `!=`
- Every occurrence of the `not` keyword was replaced with the `pNot` token
- Every occurrence of `pColonEquals` or `:=` was replaced with the `pAssignEquals` token

- Every occurrence of `=` was replaced with `==` for the comparison equals operation.

Changes made by Ethan

Overview

- Updated the block rule with new rules and removed old rules
- Removed the begin rule while maintaining the emission of begin tokens
- Merged the statement rule into the block rule to make it more generally applicable
- Added indication of public function token emission (created sPublic token)
- Maintained PT-style emission of sBegin and sEnd tokens to minimise changes needed during semantic analysis

Block Rule additions

All removed keywords were taken out of the block rule or replaced with their Quby counterparts if a direct counterpart existed (i.e. procedure → def). For new Quby keywords with no direct counterparts, corresponding rules were added.

The statement rule was removed entirely, as Quby makes no distinction between declarations and statements. The statement rule was integrated into the block rule to simplify routine parsing as a whole, so that the block rule was the primary rule crawling for the parser.

Another change to the Block rule was making sure to consume pEnd tokens, or else only the first procedure/module would be recognized.

Routine Handling changes

To simplify semantic analysis changes for Quby implementation, the parser still emits sBegin and sEnd tokens at the beginning of every block. This was implemented through modifications to the block and statement rules, so that any block of Quby will be interpreted similarly to PT Pascal in semantic analysis.

While sBegin tokens are still emitted, the BeginStmt rule for handling the actual begin keyword has been removed.

Procedure/Module handling changes

While the general form of the module rule was handled by Liam, and the procedure rule remains mostly the same as in PT Pascal, there are a couple key changes that fall under routine handling. A major change was the inclusion of the sPublic token, and subsequent changes to accomodate. For the procedure rule, changes were made within the block rule to allow for public procedures, and smeicolons were removed in the ProcedureHeading rule. For modules, due to the construction of the rule, changes were made to the module rule directly, though the form of the change was similar to that of the procedure change in the block rule.

Changes made by Noah

Overview

- Removed `repeat` statement
- Added support for `do` statement
- Modified `case` statement syntax
- Added support for `unless` statement

`repeat` Statement

This keyword was removed from the parser. To do this, the checks for the `repeat` keyword were removed from the `Block` rule along with the corresponding rule, `RepeatStmt`.

`do` Statement

The `do` statement is a purely additive addition to the parser that has the following syntax.

```
do
    % ... body of do loop

    % strictly one or more `break if` statements
    % within the body of the `do` loop are required
    break if <condition>

    % ... body of do loop
end
```

To implement this, a keyword check is performed within the `Block` rule. After which, the corresponding `DoStmt` rule is called. At the start of the rule, a `.sDo` is emitted. All statements around `break if` statements are handled by the `Block` rule. While the `break if` statement emits a `.sBreakIf` token, followed by a condition. The `do` statement is finally terminated with a `.sEnd` token.

Since this is an additive change to the language, further changes will need to be made in the semantic phase of the compiler to support this feature.

`unless` Statement

The `unless` statement is a new addition to the Quby language that has the following syntax.

```
unless <condition> then
    % body of unless statement
end
```

This behaves semantically the same as an `if not <condition>`. Since this statement is purely syntactic sugar, it can be fully implemented into Quby by only applying change to this stage of the compiler, the parser. To do this, first a check for the keyword `unless` is added for the `Block` rule along with a call to the corresponding `UnlessStmt` rule if found. This rule then emits the same contents as an `if` statements, except with the addition of the emission of an `.sNot` token at before terminating the expression.

`case` Statement

The `case` statement was a modifying change to the language which changes the syntax for the statements in Quby and adding a default branch to exit the statement with an `else`. These statement will have the following syntax.

```
case <expression>
    when <condition> then
        % `when` body
    when <condition> then
        % `when` body
    else
        % `else` body
end
```

For this statement to be syntactically valid, there must be at least 1 `when` condition in a case statement always, and that `when` statement may be accompanied 1 or more additional `when` statements and at most 1 `else` statement that acts as a default branch to exit the case statement.

This statement behaves similar to the `case` statement before but has some major changes. Firstly, the requirement for an `of` keyword after the initial expression is removed. Meanwhile, the `when` statements are purely syntactic sugar for the previous case checking syntax in PT Pascal. These statements emit the same thing as previously to the semantic phase of the compiler while consuming the token `when` before the condition and the token `then` after the condition from the scanner. The `else` branch of the new `case` statement by contract will require additional support in the semantic phase of the compiler as this feature is not currently supported. When the `else` keyword is encountered, a corresponding `.sElse` token is emitted to the next phase of the compiler, followed by the contents of the following `Block` rule called as the body of the `else` statement.

Changes made by Liam

Overview

- Added 'module' to parser
- 'while' loop updated to work with new changes to Block rule
- 'if' statement changes
- 'else' statement changes

- 'elsif' statement changes

`module` changes

Added functionality to parser to handle modules in Quby in the Block rule. First, whithin the Block rule the string 'module' is consumed then the Module rule is called. A .sModule is emited to signify that the following sIdentifier should be correlated to a module and then the Block rule is called for all subsequent declarations and statmenets to be encapsulated. The module is ended when a end is placed at the end of the declarations.

`while` loop changes

While loops were changed to fit the new changes made to ptPascal. Instead of using the Statment rule for encapsulation, the Block rule is now used. while loops still emit the same tokens as before. Handling while loops with Quby specifications will need to be done in the semantic phase.

`if` statment changes

If statmentents were changed such that they call the block rule after the expression decleration. This encapsulates the following code in sBegin and sEnd.

`else` statement changes

The else was changed so that instead of calling the statement rule, it calls the Block rule after emiting sElse.

This way the following declerations are encapsulated by sBegin and sEnd. After the Block rule, the code then exits the If rule.

`elsif` statement changes

elsif was added to the parser in a way such that it behaves as a nested if statement. Following an if statement, if an elsif follows, the code will emit an sElse, and then within the else, the If rule is called again. In the case that the if statment and subsequent elsif statment, is ended with an else statement. The elsif is nested within the if statments else. And then the else statmenet is applied to the nested if statement created by the elsif.

```
if x == 1 then
   x = 0
elsif x == 2 then
   x = 1
else
   x = 2
end
```

is the equivalent of

```
if x == 1 then
  x = 0
else
  if x == 2 then
      x = 1
  else
      x = 2
  end
end
```