CISC/CMPE 458 Phase 2

Token Definition / Other Syntactic Details

Added new Quby input tokens
Where: Parser input token list in parser.ssl and parser.pt
How: Added token definitions that had been defined in phase 1 scanner.ssl file to the input tokens of the parser.ssl file. The input tokens were added again to the parser.pt file in the same order so as the parser will make correctly.
Why: These input tokens are new to the Quby language.

Removed old input tokens
Where: Parser input token list in parser.ssl and parser.pt
How: Removed token definitions that were removed in phase 1 of the project that had been removed as output tokens of the scanner. These include tokens for the keywords not, until, program, const, procedure, begin, and repeat. The tokens are removed to maintain the same order of tokens in scanner.ssl, parser.pt, and parser.ssl
Why: These input tokens were used in PT Pascal but are no longer necessary in Quby.

Added new Quby output tokens
Where: Parser output token list in parser.ssl and parser.pt

How: Added sPublic, sModule, sDoStmt, sBreakIf, sSubstring, sLength and sIndex, and ensured the order of tokens was the same in both files.
Why: These output tokens are new to the Quby language.

Removed old output tokens
Where: Parser output token list in parser.ssl and parser.pt
How: Removed sRepeatStmt and sRepeatEnd
Why: These output tokens were used in PT Pascal but are no longer necessary in Quby.

Changes for general Syntactic Details
Where: In parser.ssl
How: The following were changes made due to the removal of old tokens and introduction of new ones: pNotEquals is now defined as !=, pNot was initialized as !, the program rule declared with a ‘using’ token rather than a ‘program’, the AssignmentOrCallStmt uses ‘=’ declaration instead of ‘:=’, expression rule no longer uses ‘<>’ instead it uses ‘!=’, the factor rule no longer uses ‘not’ and instead uses ‘!’, RepeatStmt no longer emits a .sRepeatStmt instead it outputs a .sDoStmt (‘until’ is no longer recognized neither is .sRepeatEnd, replaced with a ‘break’ choice that leads to an ‘if’ choice that outputs a .sExpnEnd token before breaking), the statement rule replaced ‘repeat’ with ‘do’, the statement rule replaced (‘const’ with ‘val’, ‘Procedure’ with ‘def’, ‘begin’ with nothing)
Why: In order to make the parser run with the new token definitions, we altered the program to use the new tokens.

Programs

Call the Block rule in the Program rule
Where: In parser.ssl
How: We removed unnecessary PT functions and calls (;) in order to suit the Quby specifications.

```
Program :
      'using'  .sProgram
      % program name

      % program parameters
      {
            pIdentifier  .sIdentifier
            [
                  | ',':
                  | *:
                  >
            ]
      }
      .sParmEnd
      @Block;
```

Why: This was done in order to allow for the Block functions to be called repeatedly while assessing multiple declarations in a Quby program.

Merge the Statements rule into the Block rule
Where: In parser.ssl
How: We merged the Statement rule into the Block rule

```
Block :
    .sBegin
      {[
            | 'val':
                  @ConstantDefinitions
            | 'type':
                  @TypeDefinitions
            | 'var':
                  @VariableDeclarations
            | 'def':
                  @DefStmt
            | 'module':
                  @ModuleStmt
            | pIdentifier:
                  @AssignmentOrCallStmt
            | 'if':
                  @IfStmt
            | 'case':
                  @CaseStmt
            | 'unless':
                  @UnlessStmt
            | 'do':
                  @DoStmt
            | 'while':
                  @WhileStmt
            | ';':
                  .sNullStmt
            | *:
                  >
      ]}
      .sEnd;
```

Why: This now allows multiple declarations and statements to be instantiated in Quby without creating a new declaration block each time.

Declarations

Removed semicolons from declarations
Where: In parser.ssl, in ConstantDefinitions, TypeDefinitions, VariableDeclarations routines
How: Removed semi colons from the parsing of constant, type and variable declarations
Why: Semicolons are not required at the end of declarations and definitions in Quby, so the output token stream must be updated.

Switched equals to colon
Where: In parser.ssl, in TypeDefinitions routine
How: Switched the = symbol to :
Why: Types are defined with colons in Quby, not with equals as in PT Pascal

Multiple variable definitions per line
Where: In parser.ssl, in VariableDeclarations routine
How: Used the following code

```
VariableDeclarations :
        % Accept one or more variable declarations.
        pIdentifier  .sIdentifier
        {[
            |',':
                pIdentifier
                .sVar
                .sIdentifier
            |*:
                >
        ]}
        ':'  @TypeBody
        ;
```

Why: In Quby, you can declare multiple variables in the same line by using commas. So we had to update VariableDeclarations to loop and check for commas and output the correct tokens as new commas and variables were declared.

Routines

Introduce recognition of 'def' in Block Rule

Where: In parser.ssl
How: By adding a def case in the Block rule

```
| 'def':
    @DefStmt
```

Why: This allows the program to recognize when a routine is being done and call the appropriate function.

Add a DefStmt to execute Routines
Where: In parser.ssl
How: By making a DefStmt that will execute a procedure and release the right tokens

```
DefStmt :
      .sProcedure
      % procedure name
      [
            | '*':
                  pIdentifier  .sIdentifier .sPublic
            | *:
                  pIdentifier  .sIdentifier
      ]
      @ProcedureHeading
      @Block
      'end';
```

Why: This allows for procedures to be generated and recognized by the compiler, where a star indicates that the routine is public. We maintain the original .sProcedure tokens in order to minimize the semantic differences.

Modules

Introduce recognition of 'module' in Block Rule
Where: In parser.ssl
How: By adding a module case in the Block rule

```
| 'module':
    @ModuleStmt
```

Why: This allows the program to recognize when a module is being done and call the appropriate function.

Add a ModuleStmt to execute Modules
Where: In parser.ssl
How: By making a ModuleStmt that will execute a module and release the right tokens

```
ModuleStmt :
      .sModule
      pIdentifier .sIdentifier
      @Block
      'end';
```
Why: This allows for modules to be generated and recognized by the compiler. We maintain the original .sModule tokens in order to minimize the semantic differences.

Statements

Adding double equals case
Where: In parser.ssl, in the expression routine
How: Adding the following case to the conditional statement
```
    | '==':
                @SimpleExpression .sEq
```
Why: Double equals is a new Quby syntax symbol that should output an equals token.

Unless Statements

Create UnlessStmt routine
Where: In parser.ssl
How: Added the following routine
```
   UnlessStmt :
        .sIfStmt
        @Expression
        .sNot
        .sExpnEnd
        'then'  .sThen
        @Statement
        [
            | 'else':
                .sElse
                @Statement
            | *:
        ];
```
Why: Unless should output a similar token stream as ifStmt, except we output sNot after sIf since unless is logically equivalent to if not.

Elsif Clauses

Create new semantic token sElsif
Where: In parser.ssl and parser.pt in the output token lists
How: Added sElsif
Why: We have chosen option 1: use a new semantic token sElsif to represent elsif.

Provide handling of elsif
Where: In parser.ssl, in the IfStmt routine
How: We added the following case to the conditional statement:

```
          | 'elsif':
                .sElsif
                @Expression
                .sExpnEnd
                'then' .sThen
                @Statement
```

Why: Similar to else, elsif is an extension of an if statement. The parser should output the new sElsif token when it appears after an if statement.

Case Statements

Introduce recognition of ‘case’ in Block Rule
Where: In parser.ssl
How: By adding a case statement in the Block rule

```
| 'case':
    @CaseStmt
```

Why: This allows the program to recognize when a case statement is being done and call the appropriate function.

Alter the CaseStmt to execute Cases
Where: In parser.ssl
How: By altering a CaseStmt that will execute a Case and release the right tokens.

```
CaseStmt :
      .sCaseStmt
      @Expression
      .sExpnEnd
      {[
            | 'when':
                  @CaseAlternative
            | *:
                  >
      ]}
      [
            | 'else':
                  .sElse @Block
            | *:
      ]
      'end'
      .sCaseEnd;
```

Why: This allows for cases to be generated and recognized by the compiler. We maintain the original .sCaseStmt tokens in order to minimize the semantic differences.

Alter the CaseAlternative to execute cases with multiple options
Where: In parser.ssl
How: By altering a CaseAlternative function that will execute an Alternative Case and release the right tokens.

```
CaseAlternative :
      {
            @OptionallySignedIntegerConstant
            [
                  | ',':
                  | *:
                  >
            ]
      }
      'then'
      @Block;
```

Why: This allows for alternative cases to be generated and recognized by the compiler. We recall the @Block function in order to allow for multiple layers of cases to exist.

Do Statements

Introduce recognition of ‘do’ in Block Rule
Where: In parser.ssl
How: By adding a do case statement in the Block rule

```
| 'do':
    @DoStmt
```
Why: This allows the program to recognize when a do statement is being done and call the appropriate function.

Add a DoStmt to execute Do statements
Where: In parser.ssl
How: By making a DoStmt that will execute a do function and release the right tokens.
```
DoStmt :
      .sDoStmt
      @Block
      'break' 'if' .sBreakIf
      @Expression
      .sExpnEnd
      @Block
      'end';
```
Why: This allows for do statements to be generated and recognized by the compiler. We maintain the original .sDoStmt tokens in order to minimize the semantic differences.

Strings

Add # choice in @Factor
Where: In parser.ssl
How: By adding a # case in the Factor rule
```
| '#':
    @Factor
    .sLength
```
Why: This allows for the Factor rule to recognize #. We maintain the original .sLength tokens in order to minimize the semantic differences.

Update the Term rule to support Quby specifications
Where: In parser.ssl
How: By altering the Term rule function to incorporate the ? choice and implement the $ rule through a different HigherFactor rule.
```
Term :
      @Factor
      @HigherFactor
      {[
            | '*':
                  @Factor  .sMultiply
                  @HigherFactor  .sMultiply
            | 'div':
                  @Factor  .sDivide
                  @HigherFactor  .sDivide
            | 'mod':
                  @Factor  .sModulus
                  @HigherFactor  .sModulus
            | 'and':
                  .sInfixAnd  @Factor  .sAnd
                  .sInfixAnd  @HigherFactor  .sAnd
            |'?':
                  @HigherFactor .sIndex
            | *:
                  >
      ]};
```
Why: This allows for the compiler to recognize and work with Quby string literal operators and output the appropriate semantic tokens.

Add a HigherFactor rule to support the Term rule

Where: In parser.ssl

How: By creating a new HigherFactor rule to specify the Quby string rule of $.

```
HigherFactor:
            @Factor
            {[
                  | '$':
                  @Factor '..' @Factor .sSubstring
                  | *:
                  >
            ]};
```
Why: This allows for the compiler to recognize and work with Quby string literal operator $ and output the appropriate semantic tokens.