Changelog Documentation

Iffy's Changes

Overview

- Updated Input Tokens
- Updated Output Tokens
- Added new Symbol Table semantic operations
- Added support for Quby Strings
  - Replaced old char-based output tokens with String
  - Added new String operation T-codes
  - Replaced char types with String
- Updated code to handle string literals as first class data types
- Updated code to handle string constants, as string variables'
- Updated code to handle string equality and inequality with specific `tStringEquals` T-code
- Updated semantic.pt with String pre-declared type and string variable allocation.

Input Token Updates

Following the previous phases, the input tokens of `semantic.ssl` where updated to match the output tokens of `parser.ssl`. They are the same order to preserve the constants assigned to the tokens by the SSL walker.

```diff
 program.

 Input :
-        % Semantic tokens - must match output tokens in parser.ssl exactly!
         sIdentifier
         firstSemanticToken = sIdentifier
         firstCompoundSemanticToken = sIdentifier
@@ -49,8 +48,6 @@ Input :
         sThen
         sElse
         sWhileStmt
-        sRepeatStmt
-        sRepeatEnd
         sEq
         sNE
         sLT
@@ -68,6 +65,22 @@ Input :
         sAnd
         sNot
         sNewLine
+
+        % added semantic tokens
+        sModule
+        sDo
+        sBreakIf
+        sSubstring
+        sLength
+        sIndex
+        sPublic
+        % end added semantic tokens
```

Output Token Updates

The old T-codes that supported the semantic behaviour for while and repeat are removed from the output tokens of `semantic.ssl` as they are not used by Quby.

```
            tCaseBegin
-           tWhileBegin
-           tRepeatBegin
-           tRepeatControl
            tCallBegin
            tParmEnd
```

```
            tCaseEnd
-           tWhileTest
-           tWhileEnd
-           tRepeatTest
            tSkipProc
            tCallEnd
```

Instead, they are replaced with new T-codes for the general `do` in

```
+           % new compund t codes
+           toDoBreakIf % compound to match ifthen t codes
+           toDoTest % compound to match while t codes
+           tDoEnd
+
+           tCaseElse % new t code for case
+                   % compound to match ifthen t codes
+           % end new compound t codes
```

A new t-code `tCaseElse` is also added to support the `else` featured added to cases in Quby.

There are other changes to the T-codes, which will be discussed in later sections.

New Symbol Table Semantic Operations

The new semantic operations `oSymbolTblStripScope` and `oSymbolTblMergeScope` were added to the `SymbolTable` semantic mechanism:

```
    oSymbolTblStripScope

    oSymbolTblMergeScope
    ;
```

Structure Updates to support Quby Strings

Changing Output Tokens

`tFetchChar`, `tAssignChar`, `tStoreChar` and `tSubscriptChar` were all replaced in the output tokens (and in their code usage) with their string counter parts in `semantic.ssl`:

```diff
        tFetchInteger
-       tFetchChar
+       tFetchString
        tFetchBoolean
        tAssignBegin
        tAssignAddress
        tAssignInteger
-       tAssignChar
+       tAssignString
        tAssignBoolean
        tStoreAddress
        tStoreInteger
-       tStoreChar
+       tStoreString
        tStoreBoolean
        tSubscriptBegin
        tSubscriptAddress
        tSubscriptInteger
-       tSubscriptChar
+       tSubscriptString
        tSubscriptBoolean
        tArrayDescriptor
```

Since Quby strings encompass both Chars and Strings, `tLiteralChar` is replaced by `tLiteralString`. Quby strings are also first-class values, and are not arrays like they were in Pascal.

New T-codes were added to support the String length, substring, index and equality operations:

```diff
        tReadEnd
+       % New output tokens
+       tConcatenate % string operators added here because same location as std operators
+                    % dont take any operand themselves, take one of the compound codes
+       tSubstring
+       tLength
+       tIndex
+       tStringEqual
+
+       tDoBegin % new item for do
+       % end new output tokens
```

Changing Semantic Mechanisms and Types

Several changes were also made to replace the semantic operations and types in `semantic.ssl` for the Pascal Char type to the Quby String.

`tpChar` was removed from the `TypeKind` type as Quby strings cover both Pascal chars and strings.

```
    type TypeKind :
            tpInteger
-           tpChar
            tpBoolean
            tpSubrange
            tpArray
```

Quby Strings are also standard primitive types, and hence replace `stdChar` in the `StdType` type:

```
    type StdType :
            stdInteger
-           stdChar
            stdBoolean
            stdText;
```

```
    type StdType :
            stdInteger
+           stdString
            stdBoolean
            stdText;
```

Both `tpString` and `stdString` replace the usage of their Char countertypes in the rules.

The `stringSize` token was added to the `Integer` type and initialized to 1024 for the size of string variables.

```
    type Integer :
            zero = 0
            one = 1
            two = 2
            three = 3
            ten = 10

            % These two sizes are machine dependent
            byteSize = 1
+           wordSize = 4
+           stringSize = 1024 % for maximum size of string
+           ;

```

`pidString` replaces `pidChar` in the `PredeclaredId` type, since Quby strings again replace the Pascal char primitive type. The usage of `pidChar` in rules was also replaced with `pidString`

```
            firstPredeclaredType = firstPredeclaredId
            pidInteger = firstPredeclaredType
-           pidChar
            pidBoolean
            pidText
            lastPredeclaredType = pidText
```

```
            firstPredeclaredType = firstPredeclaredId
            pidInteger = firstPredeclaredType
+           pidString
            pidBoolean
            pidText
            lastPredeclaredType = pidText
```

The Char trap kinds in the `TrapKind` type are also replaced with their String counterparts, initialized to the relevant codes used in the Quby runtime library.

```diff
         trReadln = 4
         trWrite = 5
         trWriteln = 6
-        trWriteString = 7
         trWriteInteger = 8
-        trWriteChar = 9
+        trWriteString = 109
         trReadInteger = 10
-        trReadChar = 11
+        trReadString = 108
         trAssign = 12;
```

They also replace any usage of the old Char trap codes in the rules.

Handling String Literals

Quby makes String first class values, unlike Pascal were Strings are treated as packed char array. This leads to changes in how a String literal is handled semantically, in the `StringLiteral` rule in `semantic.ssl`.

Now all strings are handled the same way, unlike the previous checking in Pascal:

```diff
 StringLiteral :
+        % processes a string literal, which is now only a first class value in Quby
+        % So we emit the string directly

+        oTypeStkPush(tpString)
+        oTypeStkLinkToStandardType(stdString)
+        .tLiteralString
+        oEmitString % emits the string onto it
+
+        ;
```

In simple terms:

- It pushes the type `tpString` onto the Type Stack and link that Type Stack entry to the standard type for Quby strings with `oTypeStkLinkToStandardType(stdString)`
- Next it uses emit the `tLiteralString` T-code to indicate the next token should be treated as a string
- It then uses `oEmitString` to emit the string value.

This replaces the previous method of handling strings like character arrays, removing the need for the `tSkipString` and `tStringData` T-codes, and hence why they are removed from the output T-codes:

```diff
-        tSkipString
-        tStringData
```

Handling String Constants

According to the implementation requirements, String constants should be handled as String variables.

This is done in Quby by modifying the `ConstantDefinitions` rule, after the identifier is consumed:

```
  ConstantDefinitions :           % Process named constant definitions
          {[
              | sIdentifier:
                  oSymbolStkPushLocalIdentifier
                  [ oSymbolStkChooseKind
                      | syUndefined:
                      | syExternal:
                          % A program parameter must be declared as a file variable
                          #eExternalDeclare
                      | *:
                          #eMultiplyDefined
                          % The new definition will now obscure the old one
                  ]
+
+                 % choice on the value to check for string literal
+                 [
+                     | sStringLiteral:
+                         % handle string as a var
+                         @HandleStringConstant
+                     | *:
+                         % use standard constant handling
+                         oSymbolStkSetKind(syConstant)
+                         @ConstantValue
+                         oSymbolStkEnterTypeReference
+                         oTypeStkPop
+                         oSymbolStkEnterValue
+                         oValuePop
+                         oSymbolTblEnter
+                         oSymbolStkPop
+                 ]
              | *:
```

We perform a choice operation on the value token after the identifier. If it is not a string literal, it is handled as a normal constant (`*` alternative). If it is a string literal, then we use the `HandleStringConstant` rule to implement the alternative handling:

```
+ HandleStringConstant :
+         % we handle string constants as var declaration and assignment
+         % we have consumed the identifier and the literal value
+
+         % handle the var declaration
+         % first enter the type as string, equivalent to @TypeBody rule call in VariableDeclarations
+         oTypeStkPush(tpString)
+         oTypeStkLinkToStandardType(stdString)
+
+         % use the same rule in VariableDeclarations
+         % enters the string variable into the symbol table
+         oCountPush(one) % one declaration
+         @EnterVariableAttributes
+         oCountPop % pop from the stack
+
+         % now handle the assignment part
+         % based on AssignmentStmt rule
+
+         % pop from type stack because assignment call already handles pushing type
+         oTypeStkPop
+
+         % identifier is already on top of symbol stack
+
+         .tAssignBegin
+         @Variable
+
+         % emit the string literal
+         @StringLiteral
+
+         % handle the assignment
+         .tAssignString
+
+         % then pop from the symbol stack and pop what variable pushed to type stack
+         oTypeStkPop % pops what StringLiteral put
+         oTypeStkPop % pops what Variable put
+         oSymbolStkPop
+
+
+         ;
```

The rule performs the same set of operations as done by `VariableDeclarations` and `AssignmentStmt`:

- Push the type onto the type stack
- Enter the variable attributes with `EnterVariableAttributes` rule (this also enters it into the Symbol Table)
- Pop the type from the Type Stack and the local identifier from the Symbol Stack.
- Emits `tAssignBegin` to begin an assignment statement, and uses `Variable` to get type information (makes sense since identifier is still on top of the symbol stack)
- Emits the relevant string literal
- Finishes with `tAssignString` and clearing out the Symbol stack and type stack

Implementing String Length

Implementing the String length operation involved modifying the Unary operator to add a new alternative for the `sLength` token. The Unary operator is modified since the string length operation takes one operand:

```
UnaryOperator :
        [

                oTypeStkPush(tpBoolean) % result type
                @CompareAndSwapTypes
                oTypeStkPop
+
+           | sLength:
+               .tLength
+               @HandleStringLengthOperation
            | *:
                >>
        ]
        % If an operator is present the result is an expression
        oSymbolStkSetKind(syExpression);
```

The type checking for the String length operation is handled with the `HandleStringLengthOperation` rule:

```

+ HandleStringLengthOperation:
+     % checks if it is a valid string length operation
+     % CompareAndSwapTypes is not used because the logic change required can result in semantic violations for other items
+
+     % what should be on top of the type stack should be a string
+     [ oTypeStkChooseKind
+         | tpString:
+             % if it is a string it is valid
+         | *:
+             #eTypeMismatch % it is not a string, string length operation is invalid
+     ]
+
+     % pop the type on the stack and then push result
+     oTypeStkPop
+     oTypeStkPush(tpInteger)
+     ;
+
```

The rule simply checks if the type on top of the type stack is a string, and then pops it from the stack, placing the result type: an Integer. This achieves the same functionality as the other unary operators but is specific to the string length operation.

The `CompareAndSwapTypes` rule cannot be used for type checking (unlike the other unary operators) because the String length operand takes a string type and gives an integer type as a result.

If `CompareAndSwapTypes` were to be used, we would have to make an addition to the inner choice table `tpInteger` alternative in the outer choice table, and add `tpString` as an accepted type:

```
[ oTypeStkChooseKind
    | tpInteger, tpSubrange:
        oTypeStkSwap
        [ oTypeStkChooseKind
            | tpInteger, tpSubrange:
            | tpString: % would be added here
            | *:
                #eTypeMismatch
        ]
    | tpString:
        oTypeStkSwap
        [ oTypeStkChooseKind
```

But this cannot be done as it would allow the negation of a string, which is not allowed in Quby. As a result, we define a specific rule to handle the String operation.

Handling String Equality

The String equality is handled by using the `tStringEqual` T-code rather than the `tEQ` T-code when the type on the Type Stack is a String:

```
                    @CompareOperandAndResultTypes
                | sEq:
+                   % adding support to use string equals if starting operand is string
+                   [ oTypeStkChooseKind
+                       | tpString:
+                           % If the type on type stack top is string then we are doing string comparison
+                           % emit string equal
+                           .tStringEqual
+
+                       | *:
+                           % otherwise use the standard equals
+                           .tEQ
+                   ]
                    @CompareEqualityOperandTypes
                | sNE:
```

The choice rule operates on the type on top of the type stack (which is returned by the choice operation `oTypeStkChooseKind`). If the type is a string, then `StringEqual` is emitted, otherwise `tEQ` is emitted.

String inequality is handled in a similar way, but with the emission of the `tNot` to invert the results of `tStringEqual`:

```
            | sNE:
+               [ oTypeStkChooseKind
+                   | tpString:
+                       % type is string, we have to use string inequality
+                       % can be achieved by doing inversion on equality
+                       .tStringEqual
+                       .tNot
+
+                   | *:
+                       % otherwise use standard not equals
+                       .tNE
+
+               ]
```

Handling String Substring Operation

The string substring operation is the only three-operand operation in Quby. The `TernaryOperator` rule was written to handle the substring operation and any other tri-operand operations in future:

```
TernaryOperator :
        [
            | sSubstring:
                .tSubstring
                @HandleSubstringOperandTypeChecking
                oTypeStkPush(tpString) % result is tpString

            | *:
                >>
        ]

        % If an operator is present the result is an expression
        % First pop expression types for integer operands
        oSymbolStkPop
        oSymbolStkPop
        oSymbolStkSetKind(syExpression)
    ;
```

The `TernaryOperator` rule follows a similar format to the `BinaryOperator` rule:

- It begins with a choice to see if the current semantic token is a tri-op operation, if not the rule exists
- If it is, then the appropriate t-code is emitted for the operation, followed by its type checking and then its result type is pushed onto the stack
- At the end of the rule, we pop the symbol stack twice to remove the `syExpressions` of the last 2 operands, and then set the kind of the last one to an expression
    - This makes it so that the symbol stack contains the result of the operation on top of it

It is added to the Expression rule as the other operator rules are:

```
    Expression :
            % Expressions have been converted to postfix form by the
            % previous pass with the exceptions noted below. This rule
            % pushes symbol and type table entries for the expression result.

            {
                @Operand
                @UnaryOperator
                @BinaryOperator
+               @TernaryOperator
```

The substring operation is identified by the consumption of the `sSubstring` token. When this is matched, we emit the `tSubstring` T-code, handle the type checking with the `HandleStringOperandTypeChecking` rule, and then push the result type of the operation (a string) to the type stack.

The rule `HandleStringOperandTypeChecking` is as follows:

```
HandleSubstringOperandTypeChecking:
        % check the operands
        % order on the stack would be string, int , int <top of stack>
        % so pop and check
        [ oTypeStkChooseKind % check first int operand
            | tpInteger:
                oTypeStkPop

                [ oTypeStkChooseKind % check second int operand
                    | tpInteger:
                        oTypeStkPop

                        [ oTypeStkChooseKind % check third string operand
                            | tpString:
                                oTypeStkPop
                            | *:
                                % not what we want, do error recovery
                                #eTypeMismatch
                                oTypeStkPop
                        ]

                    | *:
                        % not what we want, do error recovery
                        #eTypeMismatch
                        oTypeStkPop
                        oTypeStkPop
                ]

            | *:
                % not what we want, do error recovery
                #eTypeMismatch
                oTypeStkPop
                oTypeStkPop
                oTypeStkPop
        ]

        ;
```

This follows the format for the standard type checking. Since operands are consumed left to right, the type stack will be in the order of: string, int, int. This is followed in the type checking above: it checks for an integer type, pops the stack again and checks for an integer type and then pops the stack again to check for a string type. If the string type is matched, the type stack is popped and the rule returns.

If the types on the type stack do not match at any point, a type mismatch error is thrown and recovery is done by just popping the relevant number of types from the type stack.

Changes to `semantic.pt`

Updating token definitions

`semantic.pt` was updated with the new token constant definitions from `semantic.def` which was generated from the changes to `semantic.ssl`.

The `stdString` and `tpString` also replace the `stdChar` and `tpChar` usage in `semantic.pt`.

Updating Predeclared Types

As `pidChar` was replaced by `pidString`, its Symbol Table pre-initialization is also updated to reflect this change. The `standardCharTypeRef` was also replaced with `standardStringTypeRef` in the code:

```
            standardIntegerTypeRef: TypeTblReference;
+           standardStringTypeRef:    TypeTblReference;
            standardBooleanTypeRef: TypeTblReference;
            standardTextTypeRef:    TypeTblReference;

eam, temp, tCode, options
                  similarly for char, Boolean and text. }
                standardIntegerTypeRef := pidInteger;

+               { string }
+               symbolTblKind[pidString] := syType;
+               symbolTblTypeTblLink[pidString] := pidString;
+               typeTblKind[pidString] := tpString;
+               standardStringTypeRef := pidString;

                { Boolean }
                symbolTblKind[pidBoolean] := syType;
m, temp, tCode, options
                symbolTblKind[pidText] := syType;
                symbolTblTypeTblLink[pidText] := pidText;
                typeTblKind[pidText] := tpFile;
+               typeTblComponentLink[pidText] := standardStringTypeRef;
                standardTextTypeRef := pidText;
```

Adding String Variable Allocation

The switch case in `oAllocateVariable` was modified to allocate space for Quby Strings as they are now first class variables:

```
            oAllocateVariable:
                { Based on the structure and component entries on top
                  of the type stack
                  (structure on top). }
                case typeStkKind[typeStkTop] of
                    tpInteger, tpSubrange:
                        dataAreaEnd := dataAreaEnd + wordSize;
-                   tpChar, tpBoolean:
                        dataAreaEnd := dataAreaEnd + byteSize;
                    tpArray:
                        begin
                            assert (typeStkComponentLink[typeStkTop]
                            <> null, assert59);
```

```
            oAllocateVariable:
                { Based on the structure and component entries on top
                  of the type stack
                  (structure on top). }
                case typeStkKind[typeStkTop] of
                    tpInteger, tpSubrange:
                        dataAreaEnd := dataAreaEnd + wordSize;
+                   tpBoolean:
                        dataAreaEnd := dataAreaEnd + byteSize;
+                   tpString:
+                       dataAreaEnd := dataAreaEnd + stringSize;
                    tpArray:
                        begin
                            assert (typeStkComponentLink[typeStkTop]
                            <> null, assert59);
```

`tpChar` is removed from the case statement and a new case for `tpString` is added. The case action allocates `stringSize` bytes for each string variable, just like the Integer and subrange types.

`stringSize` is set in `semantic.ssl` (the `Integer` type) to 1024.

Ethan's Changes

Block Rule Changes

Modified the statement and block rules so that the alternatives of the statement rule are performed by the block rule, while the statement rule simply pops and pushes the scope.

Type Definition Modifications

Type definition handling has been updated to support only one per definition.

Changes to Binary Opartor Handling

The string index operator was added to the BinaryOperator rule, as well as the string concatenation operator, which was added to the sAdd section.

New Symbol Table Semantic Operations

The new semantic operations `oSymbolTblStripScope` and `oSymbolTblMergeScope` were added to the `SymbolTable` semantic mechanism. These replace the `oSymbolTblPopScope` operation from PT Pascal.

Liam's Changes

SymbolStack Assertions

Within the semantic.pt file, to allow for `syPublicProcedure` all assertions of `syProcedure` were modified to allow for assertions of `syPublicProcedure`.

New Symbol Table additions

`syModule` and `syPublicProcedure` were added to the Symbol table for identifying and treating Modules and PublicProcedures. This allows them to be added to the Symbol stack when identified.

ModuleDefinition rule

The ModuleDefinition rule was added to the rule section of Semantic.ssl. The module rule is used to consume sModule and then add the subsequent symbol to the symbol stack as a Module. Modules don't take in parameters, so we don't call the `@ProgramParameter` rule. Instead, the `@Block` rule is called to handle all declarations within the module. The oSymbolTblStripScope and oSymbolTblMergeScope are then used to promote all public symbols to the enclosing scope.

Public Procedure Handling

The Public procedure handling is done by picking up the `sPublic` output token from the semantic analizer when a procedure is being defined. Then, in the `@ProcedureDefinition` rule, we set the type of procedure it is on the symbol stack by using the case:

```
[
    | sPublic:
          oSymbolStkSetKind(syPublicProcedure)
    | *:
          oSymbolStkSetKind(syProcedure)
]
```

Modifying Variable Declaration

The `VariableDecleration` and `EnterVariableAttributes` rules were changed to allow for multiple identifiers declared using one type.

In the `VariableDeclaration` rule, a loop is used to increment a counter for each subsequent `sVar` emitted by the parser. Each of the new variables has its identifier pushed to the symbol stack.

Then the rule calls the `EnterVariableAttributes` rule, which we now loop through for the amount which the counter holds, while modifying the Symbol and Type stacks accordingly.

```
{[oCountChoose %Loop through the count and decrement per variable declaration
            | zero: %If no more variable declerations exit
                >
            | *:
                oCountDecrement

                // Rest of unmodified EnterVariableCode
]}
```

The above looping was also used to pop from the symbol stack at the end of the variable declerations.

Noah's Changes

Overview

- Removed support for `repeat` statement
- Added support for `do` loop
- Added support for `else` statement in `case`
- Removed support for multiple constant definitions in one line

`repeat` statement

This change does not have much involved with it. All that was required here was deleting artifacts of the RepeatStmt rule and its detection in 'semantic.ssl'.

`do` loop

This change primarily required adding in a new rule, `DoStmt`. First, in the `Block` rule, the `DoStmt` rule is called upon detection of an `sDo` token. Upon this the rule continually allows for either regular statements with the `Statement` rule and interspersed `break if` statements that will be bounded by the T-codes `tDoBreakIf` and `tDoTest`. Finally, the loop is terminated with a `tDoEnd` token.

`case` statement

The majority of the `case` statement handling from PTPascal is used again in this phase. This is because in the previous phase of the compiler the body of the statement (with exception of the `else`) is emitted in a way to mimick the output of that of PTPascal. To handle the new `else` statement, a loop is added to the rule to check for regular case alternative and `else` statements, as well as an `sCaseEnd` token to terminate the statement. Upon detection of the `sElse` token, the `.tCaseEnd` token is emitted so the body of the case statement can be emitted exactly as before. After this, the else statement is emitted and a branch added in the case statement, ending in the consumption of the `sCaseEnd` token from the previous step. Thus, effectively making the `else` statement seem separate from the `case` statement making its handling more similar to that previously done in PTPascal.

Multiple constant definitions

This change was a relatively simplistic one as there exists a rule already called `ConstantDefinitions` which was promptly renamed to `ConstantDefinition` now. in the rule before, a loop was used to gather all the declared constants. To support only a single definition, this loop was removed and replaced with its body, so now only a single identifier and subsequent assignment is accepted.