Phase 3 Semantic Analyzer Documentation

Table of Contents

Statements and Declarations

Merged Statements into Block
Where: In semantic.ssl, in Block rule
How: Copied the conditionals from statements and pasted them into block.
Why: Quby allows for the arbitrary mixing of declarations and statements

Modify Block
Where: In semantic.ssl, in Block rule
How: Removed sBegin handling within the loop, removed sBegin handling at the end of loop. Added sBegin at beginning and sEnd at end of rule.
Why: Quby allows for the arbitrary mixing of declarations and statements

Modify Statements
Where: In semantic.ssl, in Statements
How:

```
Statement :
        % Push a new scope
        oSymbolTblPushScope
        @Block
        % Pop the scope
        oSymbolTblPopScope;
```

Why: Quby allows for the arbitrary mixing of declarations and statements

Modify handling of constant definitions
Where: In semantic.ssl, in ConstantDefinitions
How: Removed the loop by removing { } and also removed >
Why: Quby only allows one constant definition at a time.

Modify handling of type definitions
Where: In semantic.ssl, in TypeDefinitions
How: Removed the loop by removing { } and also removed >
Why: Quby only allows one type definition at a time.

Modify handling of variable declarations
Where: In semantic.ssl, in VariableDeclarations
How:

```
VariableDeclarations :
        sIdentifier
        oSymbolStkPushLocalIdentifier
        [ oSymbolStkChooseKind
            | syUndefined, syExternal:
            | *:
              #eMultiplyDefined
        ]
        {[
            | sVar:
              sIdentifier
              oSymbolStkPushLocalIdentifier
              oCountIncrement
              [ oSymbolStkChooseKind
                  | syUndefined, syExternal:
                  | *:
                    #eMultiplyDefined
              ]
            | *:
              >
        ]}
        @TypeBody
        {[ oCountChoose
            | zero:
              >
            | *:
              @EnterVariableAttributes
              oSymbolStkPop
              oCountDecrement
        ]}
        oTypeStkPop
        oCountPop;
```

Why: Quby allows multiple identifiers to be declared using one type, but only one declaration per definition.

Modules

Added ModuleDefinition
Where: In semantic.ssl
How: Added ModuleDefinition to Block rule and created the following rule:

```
ModuleDefinition :
        sIdentifier  oSymbolStkPushIdentifier
        [ oSymbolStkChooseKind
            | syUndefined:
            | syExternal:
                #eExternalDeclare
            | *:
                #eMultiplyDefined
        ]
        oSymbolStkSetKind(syModule)
        oTypeStkPush(tpNull)
        oTypeTblEnter
        oSymbolStkEnterTypeReference
        oSymbolTblEnter
        oSymbolTblPushScope
        @Block
        oTypeTblUpdate
        oTypeStkPop
        oSymbolTblUpdate
        oSymbolStkPop
        oSymbolTblStripScope
        oSymbolTblMergeScope;
```

Why: Added rule for modules that would be called in Block when a module is used.

Changed ProcedureDefinition

Where: In semantic.ssl

How: Added sPublic option to proceduresDefinition with syPublicProcedure:

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

Why: If module is public it outputs PublicProcedure instead of syProcedure.

Added New Semantic Mechanism

Where: In semantic.ssl

How: Added oSymbolTblStripScope and oSymbolTblMergeScope to SymbolTable:

```
oSymbolTblStripScope

oSymbolTblMergeScope
```

Why: Adding tables to support public ModuleDefinition

String

Alter Char References to String References
Where: semantic.pt and semantic.ssl
How:

```
semantic.pt

{ Input / Output Tokens }
tFetchString = 24;
tAssignString = 29;
tStoreString = 33;
tSubscriptString = 38;
eStringExpnReqd = 34;
pidString = 2;
stdString = 1;
tpString = 1;
trWriteString = 109;
trReadString = 108;

{ Predefined type table entries }
standardStringTypeRef: TypeTblReference;

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

{ text (i.e. file of char) }
typeTblComponentLink[pidText] := standardStringTypeRef;

procedure Error (errCode: ErrorCodes);
eStringExpnReqd:
write('str type expression required');

oTypeStkLinkToStandardType:
stdString:
typeStkTypeTblRef[typeStkTop] := standardStringTypeRef;
```

```
      oValuePushString:

semantic.ssl

Input :
tFetchString
tAssignString
tStoreString
tSubscriptString
tLiteralString

Error :
eStringExpnReqd

type PredeclaredId :
      pidString

type StdType :
      stdString

type TypeKind :
      tpString

type TrapKind :
      trWriteString = 109
      trReadString = 108

mechanism Emit :
      oValuePushString

AllocateVar :
      | tpString, tpBoolean:

EmitAssign:
      | tpString:
            .tAssignString

EmitStore:
      | tpString:
            .tStoreString

CompareAndSwapTypes :
      | tpString:
```

```
            [ oTypeStkChooseKind
                  | tpString:

CompareRelationalOperandTypes :
      | tpString:
            [ oTypeStkChooseKind
                  | tpString:
CompareEqualityOperandTypes :
      | tpString:
            | tpString:

VariableExtension :
      | tpString:
            .tSubscriptString

VariableOperand :
      | tpString:
            .tFetchString
            oEmitDataAddress

oTypeStkPushSymbol
      oTypeStkPush(tpString)
      | rtOrd:
            | tpSubrange:
                  #eStringExpnReqd

AssignProcedure :
      [ oTypeStkChooseKind
            | tpString:

      .tSubscriptString

      | tpString, tpArray:

WriteProcedure :
      | sIdentifier:
            | tpString:

WriteText :
      [ oTypeStkChooseKind
            | tpArray:
                  | tpString:
```

```
ReadProcedure :
      | sIdentifier:
            [ oTypeStkChooseKind
                  [ oSymbolStkChooseKind
                        | tpString:

ReadText :
      [ oTypeStkChooseKind
            | tpString:
                  oEmitTrapKind(trReadString)

extFileVariable :
      | tpFile:
            | syVariable, syVarParameter:
                  | tpString:

SymbolStkPushDefaultStringConstant :
      oTypeStkPush(tpString)
oTypeStkLinkToStandardType(stdString)
```

Why: In order for the compiler to analyze string literals rather than the old PT characters. The above changes were made to translate the program to use only String type variables and remove all references to characters.

Alter StringLiteral Rule:
Where: semantic.ssl
How:

```
Operand :
      | sStringLiteral:
            oValuePushString
oSymbolStkPush(syExpression)
oTypeStkPush(tpString)
oTypeStkLinkToStandardType(stdString)
@StringLiteral

StringLiteral :
      .tLiteralString
      oValuePushStringLength
      oEmitValue
      oValuePop
      oEmitValue
      oValuePop;
```

Why: The string rule no longer emits a packed array like it did with the old PT compiler. To satisfy the Quby language, we now emit only the length and the string literal itself.

Implement sLength UnaryOperator
Where: semantic.ssl
How:

```
UnaryOperator :
      | sLength:
                  .tLength
                  oTypeStkPush(tpInteger) % result type
                  @CompareAndSwapTypes
                  oTypeStkPop
```

Why: This function sLength is an operator that allows for the emission of string lengths as per Quby specifications.

Implement sConcatenation, sEqual, sIndex BinaryOperator
Where: semantic.ssl
How:

```
BinaryOperator :
      | sAdd:
                  [ oTypeStkChooseKind
                  | tpInteger:
                        .tAdd
                        oTypeStkPush(tpInteger) % result type
                        @CompareOperandAndResultTypes
                  | tpString:
                        .tConcatenate
                        oTypeStkPush(tpString) % result type
                        @CompareOperandAndResultTypes
                  | *:
                        >>
                  ]

      | sEq:
                  [ oTypeStkChooseKind
                  | tpInteger:
                        .tEQ
                  | tpString:
                        .tStringEqual
                  | *:
                        >>
```

```
                ]
                @CompareEqualityOperandTypes

       | sNE:
                [ oTypeStkChooseKind
                    | tpInteger:
                        .tNE
                    | tpString:
                        .tStringEqual
                        .tNot
                    | *:
                        >>
                ]
                @CompareEqualityOperandTypes

      | sIndex:
                .tIndex
                oTypeStkPush(tpString)
                @CompareOperandAndResultTypes
                oTypeStkPop
                oTypeStkPush(tpInteger)
```

Why: tConcatenate and tStringEqual are both emitted when operations are made in Quby to trigger those commands. Since strings are treated like integers in Quby, they act in the same way as two integers being added, thus we placed the concatenation behavior with integer addition. In the same way, strings can be equated in Quby and that is done in sEq and SNE where we release the appropriate t-codes.

Implement Ternary Operator Rule

Where: semantic.ssl

How:

```
Expression :
      @TernaryOperator

TernaryOperator:
      [
            | sSubstring:
                [oTypeStkChooseKind
                    | tpString:
                    | *:
                        #eTypeMismatch
                ]
```

```
                oTypeStkPop
                oTypeStkPush(tpString)
                .tSubstring
            | *:
        ];
```

Why: Substrings is an operation that can be done in Quby. We implemented it in Ternary Operator rule to ensure its function.

Add T-Code String Operator Tokens

Where: semantic.ssl

How:

```
tConcatenate
tSubstring
tLength
tIndex
tStringEqual
```

Why: These t-codes are necessary for the implementation of string literal related operations.

Add String Size Parameter

Where: semantic.pt and semantic.ssl

How:

```
semantic.pt

stringSize = 1024;

oAllocateVariable:
tpString, tpBoolean:
dataAreaEnd := dataAreaEnd + stringSize;
semantic.ssl

type Integer :
      stringSize = 1024;
```

Why: Since characters no longer dictate strings in Quby, we implemented setting the new sizes for strings as 1024 as per Quby specifications.

Implement Strip Scope Mechanism

Where: semantic.pt

How:

```
semantic.pt

oSymbolTblStripScope:
```

```
      { Pop the lexic level stack, remove local entries from the type
table,
      remove local entries but leave parameter entries on the symbol stack.
      But do not decrement lexical levels nor change symbolTblTop,
typeTableTop. }

      begin
      Assert((lexicLevelStackTop >= 1), assert31);
      i := symbolTblTop;
      { Set the identifier table pointer to the identifier entry in the
            closest enclosing scope if there is one. }
      while i > symbolTblDisplay[lexicLevelStackTop] do
            begin
                  link := symbolTblIdentLink[i];

                  if link <> null then
                  { This is not a dummy identifier generated by
                        the parser's syntax error recovery procedure. }
                  begin
                        while link > 0 do
                              link := symbolTblIdentLink[link];
                        identSymbolTblRef[-link] := symbolTblIdentLink[i];
                  end;

                  i := i - 1
            end;
      end;
```

```
semantic.ssl

oSymbolTblStripScope
```

Why: StripScope is a new operation in Quby that allows for the popping of the lexical stack, removing local entries from the type table, but without decrementing the lexical levels nor changing the symbleTblTop/typeTableTop as per Quby specifications.

Implement Merge Scope Mechanism

Where: semantic.pt

How:

```
semantic.pt

oSymbolTblMergeScope:
```

```
      Begin
            Assert((lexicLevelStackTop >= 1), assert31);
            lexicLevelStackTop := lexicLevelStackTop - 1;
      end;

semantic.ssl

oSymbolTblMergeScope
```

Why: MergeScope is a new operation in Quby that allows for decrementation of the lexical top stack as per Quby specifications.

Alter string constants to act like vars

Where: semantic.ssl

How:

```
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
        ]
        oSymbolStkSetKind(syConstant)
        @ConstantValue
        oSymbolStkEnterTypeReference
        oSymbolStkEnterValue
        oValuePop
        oTypeStkPop
        oSymbolTblEnter
        oSymbolStkPop;
```

```
ConstantValue :         % push the value and type of the constant value
        [
...
                        ]
                    | *:
                ]
```

```
            oSymbolStkPop
            oSymbolStkPop
        | sStringLiteral:
            .tAssignBegin
            .tLiteralAddress
            oAllocateAlignOnWord
            oSymbolStkEnterDataAddress
            oEmitDataAddress
            oAllocateVariable
            oTypeStkPop
            @StringLiteral  % pushes type and value, enters address
            oSymbolStkSetKind(syVariable)
            .tAssignString
    ];
```

Why: In Quby, string constants need to behave like variable declarations. This can be done by altering the ConstantDefinitions Rule.

Elsif Clause

Modified IfStmt
Where: In semantic.ssl, in IfStmt
How: Add case for sElsif as follows

```
IfStmt :
        .tIfBegin
        @BooleanControlExpression
        sThen  .tIfThen
        oFixPushForwardBranch
        oEmitNullAddress                % false branch
        @Statement
        [
            | sElse:
              .tIfMerge
              oFixPushForwardBranch
              oEmitNullAddress                % true branch
              oFixSwap                % false branch back on top
              oFixPopForwardBranch
              @Statement
            | sElsif:
              .tIfMerge
              oFixPushForwardBranch
              oEmitNullAddress                % true branch
              oFixSwap                % false branch back on top
              oFixPopForwardBranch
              @IfStmt
            | *:
        ]
        .tIfEnd
        oFixPopForwardBranch;
```

Why: Semantic analyzer must handle the new sElsif semantic token.

Do Statement

Remove handling of PT RepeatStmt
Where: In semantic.ssl
How: Deleted RepeatStmt, in Block rule, SDoStmt now calls DoStmt
Why: No longer need handling of PT RepeatStmt

Created DoStmt
Where: In semantic.ssl
How:

```
DoStmt :
        .tDoBegin
        oFixPushTargetAddress           % top-of-loop branch target
        @Statement
        sBreakIf .tDoBreakIf
        @BooleanControlExpression
        .tDoTest
        oFixPushForwardBranch
        oEmitNullAddress                % exit branch
        oFixSwap                % top-of-loop target back on top
        @Statement
        .tDoEnd
        oFixPopTargetAddress
        oFixPopForwardBranch;
```

Why: Need handling of do statements. Similar logic to while statement but statement sequence is allowed before the break if part.

Modified tokens
Where: In token list in semantic.ssl
How: Changed tRepeatBegin to tDoBegin, deleted repeatControl, changed tRepeatTest to TDoTest, added TDoEnd, addedtDoBreakIf
Why: Need to update tokens to support transition from repeat to do.

Modified tokens
Where: In token list in semantic.pt
How: Aligned to token list in semantic.ssl
Why: Need to update tokens to support transition from repeat to do.

Case Statement Else Clause

Modified CaseStmt
Where: In semantic.ssl, in CaseStmt
How:

```
CaseStmt :
        .tCaseBegin
        @CaseSelectorExpression
        oCasePushDisplay        % handle nested case statements
        oCountPush (zero)       % count case alternative statements
        .tCaseSelect
        oFixPushForwardBranch
        oEmitNullAddress        % address of case branch table
        {[
            | sCaseEnd:
                .tCaseEnd
                oFixPopForwardBranch
                oEmitCaseBranchTable
                >
            | sElse:
                .tCaseEnd
                oFixPopForwardBranch
                oEmitCaseBranchTable
                @CaseElse
                sCaseEnd
                >
            | *:
                @CaseAlternative
        ]}

        % emit merge branches for case alternatives
        {[ oCountChoose         % number of case alternatives
            | zero:
                >
            | *:
                oFixPopForwardBranch
                oCountDecrement
        ]}
        oCasePopDisplay
        oCountPop;
```

Why: Need to support handling of else clause in case statements.

Created CaseElse
Where: In semantic.ssl, in CaseElse
How:

```
CaseElse:
    %adding this to address case else
    .tCaseElse
    @Statement
    .tCaseMerge
    oFixPushForwardBranch
    oEmitNullAddress                % merge branch at end of statement
    oFixSwap                        % keep case select branch on top
    oCountIncrement;
```

Why: Supports CaseStmt handling of else clause in case statements.

Created tCaseElse token

Where: In semantic.ssl and semantic.pt

How: Added tCaseElse in list of non-compound tokens

Why: Add support for tCaseElse token that is emitted in CaseElse